IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 12 May 2025, accepted 11 July 2025, date of publication 16 July 2025, date of current version 23 July 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3589802

RESEARCH ARTICLE

MLCRP: ML-Based GPU Cache Performance Modeling Featured With Reuse Profiles

MINJUNG CHO AND EUI-YOUNG CHUNG, (Member, IEEE)
Department of Electrical and Electronic Engineering, Yonsei University, Seoul 03722, Republic of Korea

Corresponding author: Eui-Young Chung (eychung@yonsei.ac.kr)

This work was supported in part by the Technology Innovation Program through Ministry of Trade, Industry and Energy of Korea, under Grant RS-2024-00420541 and Grant 2410000802; and in part by the National Research Foundation of Korea (NRF) through Korea Government [Ministry of Science and Information and Communication Technology (MSIT)] (Plug and Play (P&P) Chiplet Integration Research Center) under Grant RS-2024-00405495.

ABSTRACT Accurate cache performance prediction is critical for designing efficient memory hierarchies in high-performance computing systems. While cyclic simulators provide high accuracy, they require significant computational cost and time, making them inefficient for large-scale design space exploration. Analytical models are faster but lack accuracy in complex cache scenarios. This paper proposes MLCRP, a machine learning-based GPU cache performance prediction framework that utilizes the reuse profile (RP) as a key feature. RP captures memory access locality through a histogram of reuse distances. MLCRP consists of three main stages: data preparation, training, and inference. In the data preparation stage, synthetic RP-based traces are generated from parameterized distributions to simulate diverse and non-stationary memory patterns. In the training stage, a regression-based ML model is trained to capture the relationship between RP features, cache configurations, and performance metrics such as miss rate and miss status holding register (MSHR) merge rate. Finally, we propose a method to extract RP features from real GPU application traces, enabling the trained model to predict cache performance. Experimental results demonstrate that MLCRP significantly improves prediction accuracy compared to existing analytical models, maintaining the mean absolute error (MAE) within 5%. Furthermore, it successfully reduces simulation time by an average of four orders of magnitude compared to cycle-accurate simulators. Combining the strengths of analytic speed and simulation accuracy, MLCRP offers a scalable and generalizable solution for GPU cache modeling.

INDEX TERMS Cache memory, reuse distance, reuse profile, machine learning, train data generation.

I. INTRODUCTION

In high-performance computing architectures, the GPU memory systems are playing increasingly critical roles. In particular, cache memory significantly impacts processor performance and power efficiency, prompting active research to optimize it [1], [2]. Consequently, while cache performance modeling has traditionally been a prominent research area, its significance has been further highlighted.

The need for cache performance modeling arises from two primary factors. First, advancements in computer architecture have led to a substantial increase in the scale and complexity of cache systems. Exploring the vast design space to find optimal cache configurations requires considering both hardware and application characteristics. Second, while cycle-accurate simulators such as Gem5 [3], GPGPU-Sim [4], and Accel-Sim [5] provide high accuracy, their slow simulation speeds are a limiting factor. As the number of cache configurations to explore increases, the time required for simulation grows exponentially, making it practically impossible to evaluate all potential designs exhaustively. Thus, cache performance modeling plays a critical role in addressing these limitations.

Cache performance modeling helps reduce the time and computational resources needed for simulations. Such methodologies [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21] not only maximize

The associate editor coordinating the review of this manuscript and approving it for publication was Stavros Souravlas.

design efficiency but also facilitate effective design space exploration (DSE).

Two primary approaches have been widely studied for cache performance modeling and DSE: analytic models and simulation-based models. The first approach involves developing performance analytic models [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], which are based on theoretical models or mathematical analyses using reuse distance (RD) or reuse profiles (RP). RD is a critical locality metric used to evaluate performance indicators [22]. RP is data that represent the frequency of RD and can also be utilized for predicting cache performance. Simple models are highly advantageous due to their low computational overhead but face challenges in accuracy as the target cache model grows more complex.

The second approach seeks to overcome these limitations by accelerating simulation or mimicking cache behavior [17], [18], [19], [20], [21]. These methods model only the cache or partially simulate it, enhancing runtime efficiency. However, they face fundamental constraints in time efficiency due to the vast design space. Analytic models are fast but less accurate, while simulation-based methods are highly accurate but slower.

Machine learning (ML) techniques have demonstrated success in system-level modeling [23], [24], [25], but ML frameworks specifically tailored for GPU cache performance modeling are still underdeveloped. Moreover, building an effective ML-based cache predictor requires addressing two critical challenges: (1) how to construct meaningful input features that reflect GPU memory locality, and (2) how to build diverse and representative datasets that ensure generalization across cache configurations and workloads.

To this end, we propose two key strategies tailored to GPU cache memory systems. First, we use RP data as the primary input feature. RP represents the frequency distribution of reuse distances and provides a rich description of memory access locality in GPU workloads. Second, we introduce a data generation methodology that creates synthetic RP-based training data by segmenting traces into fixed-length intervals. This accounts for the non-stationary nature of memory accesses in GPUs and improves model training and robustness.

In this paper, we introduce **MLCRP**, an ML-based approach to GPU cache performance modeling featured with RP data. MLCRP consists of three phases: **Data Preparation**, **Training**, and **Inference**.

In the data preparation phase, training data is generated based on RP data. To account for non-stationary data characteristics, RP data is divided into fixed intervals to generate synthetic data, and extracted from various cache configurations. In the training phase, the ML-based cache performance model is trained using the defined input features (represented by RP) and output labels (cache miss rates and miss status holding register (MSHR) merge rates). Finally, in the inference phase, the trained model predicts cache performance, with methods proposed for extracting features from real applications. The cache prediction performance of MLCRP is analyzed by comparing its results with those of cycle-accurate simulators across various applications.

TABLE 1. Example of memory references and RD.

Time	0	1	2	3	4	5	6	7	8	9
Addr	a	b	c	a	a	c	b	d	a	a
RD	-1	-1	-1	2	0	1	2	-1	3	0

RD	#	Freq.
-1	4	0.4
0	2	0.2
1	1	0.1
2	2	0.2
3	1	0.1

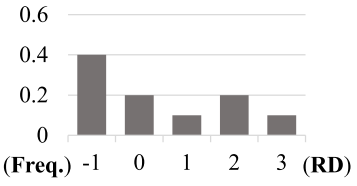

FIGURE 1. RP calculation for the Table 1.

To the best of our knowledge, MLCRP is the first machine learning-based framework specifically designed to predict GPU cache performance metrics—such as miss rates and MSHR merge rates—using RP features. The key contributions of this paper are summarized as follows:

- We propose MLCRP, an ML-based GPU cache performance modeling framework that maintains the efficiency of analytic approaches while achieving accuracy comparable to simulation-based models.
- MLCRP leverages RP features as expressive inputs that enable accurate prediction of GPU cache performance metrics such as cache miss rate and MSHR merge rate. Notably, the MSHR merge rate is difficult to estimate using traditional models.
- MLCRP introduces a practical and scalable data generation methodology that segments memory traces to capture non-stationary behavior, thereby improving training efficiency, and increasing dataset diversity.

The remainder of this paper is organized as follows: Section II reviews the background and related work. Section III outlines our motivation for this study. Section IV introduces the proposed MLCRP framework in detail. Section V presents the experimental results, and Section VI concludes the paper with a discussion of the findings.

II. BACKGROUND AND RELATED WORK

Reuse distance (RD) and its histogram-based extension, reuse profile (RP), are widely used to model memory access locality and predict cache performance [22], [26]. Based on these metrics, prior cache modeling techniques are generally divided into two categories: analytic models, and simulation-based models. This section reviews RD/RP definitions and summarizes related work across both modeling paradigms.

A. REUSE DISTANCE AND REUSE PROFILE

RD represents the difference in the number of unique memory addresses between two consecutive accesses to the same

TABLE 2. Comparison of different cache modeling methods for *AlexNet*.

Method	Input Data Format (size)	Cache Performance (miss/merge)	Run Time (seconds)	Drawback
Simulation (Accel-Sim)	App trace (21.65GB)	0.081 / 0.164	1.89×10^5	Very slow
Analytic (PPT-GPU-Mem [15])	RP data (6.20MB)	0.071 / -	1.05	Low accuracy; stationary RP data; no merge rate

memory reference. RD is typically used to predict cache misses based on the 3C model (Compulsory, Capacity, and Conflict misses). The RD for the first access to an address is denoted as -1 or ∞, and in Table 1, it is represented as -1. An RD of -1 for memory access indicates that the data does not exist in the cache, leading to a compulsory miss.

As shown in Table 1, when memory addresses a, b, c, and d are accessed sequentially, the RD is defined as -1 for the first access to each memory address, and RD is calculated for each subsequent access. At times 0, 1, and 2, addresses a, b, and c are accessed for the first time, resulting in an RD of -1. At time 3, when an address a is accessed again, the RD becomes 2, representing the number of unique addresses between the first access at time 0 and the current access. Similarly, at time 4, when an address a is accessed again, the RD is 0. At time 5, when address c is accessed, the RD is 1, reflecting the unique addresses between the current access and the previous access at time 2. This process is repeated for the entire length of the trace, and the length of the RD is the same as the length of the trace.

A histogram representing the frequency of each RD is called the RP. Fig. 1 illustrates the RP for Table 1, showing the frequency of each RD value divided by the total number of memory accesses. The RP is hardware-independent and, once extracted from a memory reference set, can be analyzed for various cache hardware configurations.

B. ANALYTIC CACHE PERFORMANCE ANALYSIS

Analytical approaches [27], [28] offer faster computation by relying on heuristics, but they are limited in accuracy. These methods typically involve extracting and analyzing trace data. Early approaches required large-scale input data provided by compilers or profilers, making them inefficient for DSE [29]. Additionally, micro-benchmarking methods used to extract hardware parameters were limited because they only considered two applications [30]. In CPUs, techniques for deriving RP by modeling data sharing have been proposed [31], as well as the introduction of concurrent RD [32]. However, these methods faced hardware-dependent issues. Further research explored cache architecture design using private RP [33], [34], resulting in analytical models that estimate average memory access time based on RP. PPT-GPU-Mem [15] introduced a method for extracting RP once based on hardware, but it had limitations, as it focused only on cache miss rate. Although it was proposed as a faster and more accurate parameterized GPU simulator compared to GPGPU-Sim, its simplified equations were insufficient for fully predicting cache performance [35].

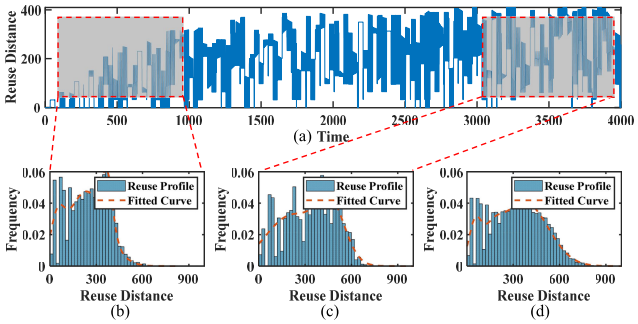

FIGURE 2. RD data and RP data for *AlexNet*: (a) RD data, (b) RP data between time 0 and 1000, (c) RP data between time 3000 and 4000, and (d) RP data over the entire time period.

C. SIMULATION-BASED CACHE PERFORMANCE ANALYSIS

Cycle-accurate simulators, including Gem5 [3], GPGPU-Sim [4], and Accel-Sim [5], provide detailed results for cache hits and misses. However, their primary drawback is long simulation times. Various methods have been proposed to reduce simulation time for efficient cache DSE [17], [18], but they still failed to efficiently explore a wide range of cache configurations. RD has been used to analyze L1 cache performance [14] and memory access patterns in applications [36]. One major challenge in collecting memory access information is the lengthy process of extracting the RP. Simulation-based predictive models for kernels have also been suggested, but they are unsuitable for DSE because the reuse distances must be recalculated every time the cache configuration changes [20], [21]. This approach was limited to calculating the RP for a single thread block and struggled to generalize it to overall performance [20]. Additionally, DRDA has been employed to investigate the impact of MSHRs and various cache configurations using RD [21], but it had the limitation of requiring RD reconstruction for each new cache configuration.

Some memory accesses that result in misses can be merged if they target the same block, reducing downstream traffic and improving effective bandwidth. Since MSHRs significantly affect cache performance [20], they must be accounted for when analyzing cache performance using RD.

Therefore, a new approach is needed that retains the fast computation times of analytical methods while offering higher accuracy.

III. MOTIVATION

The results of cache performance modeling for real workloads are presented in Table 2. The image classification

network, *AlexNet*, was evaluated using two different cache modeling methods. The performance analysis was conducted with an L1 cache configured with 8 sets and 8 ways per SM. Simulation results were obtained using a trace-driven simulator, Accel-Sim [5], and the analytical model employed was PPT-GPU-Mem [15]. These two models adopt distinct approaches and input data formats for cache performance evaluation.

A. INPUT DATA FORMAT

The two models utilize different input data formats. The simulation approach uses application traces as input, whereas the analytical model relies on RP data. The input data for the analytical model is extracted from the simulation input (application trace) by generating RD data, from which frequencies are calculated to produce RP data. Consequently, the data size is reduced from 21.65 GB for the application trace to 6.20 MB for the RP data—a reduction of approximately 1000 times.

Based on the use of RP data for performance modeling in the analytical model, we propose leveraging RP data as features in a machine learning model. However, using RP data as-is has limitations because RP data preserves only the stationary characteristics of the application.

Fig. 2 illustrates (a) RD traces and (b)-(d) RP data for *AlexNet*. Fig. 2(b) shows RP data from time 0 to 1000, Fig. 2(c) covers time 3000 to 4000, and Fig. 2(d) represents RP data for the entire time range. However, as Fig. 2(d) illustrates, RP data extracted from the entire trace tends to average out temporal variations, capturing only stationary characteristics. This fails to reflect dynamic changes observed in specific intervals such as Fig. 2(b) and 2(c), highlighting the need for temporally segmented, non-stationary RP inputs.

1) CACHE PERFORMANCE

Compared to simulator results, the analytical model demonstrates lower accuracy and does not provide information about the MSHR merge ratio. However, when comparing execution times, the analytical approach is is over five orders of magnitude faster than simulation. In summary, the analytical approach offers high speed but limited accuracy. Hence, a new modeling method is required to enhance prediction accuracy, integrate MSHR merge ratio predictions, and reduce execution time.

To overcome these limitations, it is necessary to generate training data that reflects the non-stationary characteristics of input data by utilizing RP information as features and trace information. Applying this approach to machine learning-based cache performance modeling is expected to develop a model that reduces execution time while predicting performance more accurately.

In summary, we aim to bridge the gap between accuracy and efficiency in GPU cache modeling by using machine learning on temporally segmented RP features, enabling scalable and fine-grained performance prediction.

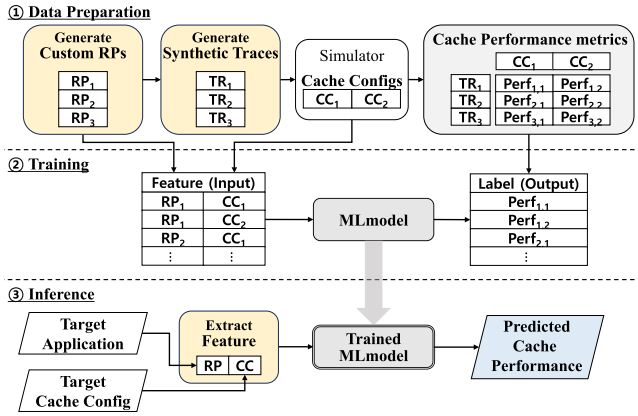

FIGURE 3. Overview of MLCRP, consisting of the Train, Inference, and evaluation phases.

IV. MLCRP: ML-BASED GPU CACHE PERFORMANCE MODELING FEATURED WITH REUSE PROFILES

A. OVERVIEW

We propose MLCRP, a machine learning-based GPU cache performance modeling framework that utilizes RP to predict cache behavior. As shown in Fig. 3, MLCRP is composed of three main stages: data preparation, training, and inference.

In the data preparation stage, diverse RP-based training data is generated to reflect various memory access patterns and cache configurations. In the training stage, machine learning models learn the relationship between RP features and key cache metrics such as miss rate and MSHR merge rate. In the inference stage, RP is extracted from a target application and used to estimate cache performance under given configurations.

The following subsections detail each component of the framework.

B. DATA PREPARATION STAGE

As shown in the first part of Fig. 3, the data preparation stage consists of two main tasks. The first task is generating RP data, which is utilized as a feature. To enhance the diversity of the training data, a specific method for generating RP data is proposed. The second task involves creating synthetic traces using the generated RP data. These synthetic traces are generated with varying RPs at regular intervals to mimic the non-stationarity observed in real-world applications. Finally, the synthetic traces are employed to extract cache performance metrics and prepare the training dataset.

1) GENERATING CUSTOM RPs

Figs. 2(b)-2(d) show that real RP data often exhibits multimodal distributions that can be approximated by mixtures of Gaussians. These distributions arise from the spatial locality and varying access phases of applications. Based on this observation, we generate synthetic RPs by sampling N Gaussian peaks across a defined RD range $[0, MaxRD]$. The

Algorithm 1 Generate Custom RPs

1: **Input:** N, $MaxRD$
2: **Output: RP** $= \{(rd_i, freq_i) \mid i = 0, 1, \ldots, M\}$
3: Step 1. Set positions of Gaussian distributions:
4: $n \leftarrow 0$
5: **while** $n < N$ **do**
6: $\quad pos_n = RND_i(0, MaxRD)$
7: $\quad ratio_n = RND_f(0, 1)$
8: $\quad n \leftarrow n + 1$
9: **end while**
10: $pos_N = -1$
11: $ratio_N = RND_f(0, 1)$
12: normalize **ratio**
13: Step 2. Generate Gaussian distributions:
14: $i \leftarrow 0$
15: **while** $i < M$ **do**
16: $\quad rd_i = i$
17: $\quad freq_i = \text{sum}(ratio_n \cdot gaussian(rd_i, pos_n, \sigma))$
$\quad for \;\; n = 0, 1, \ldots, N,$ and $\sigma \in (0, 2)$
18: $\quad i \leftarrow i + 1$
19: **end while**
20: $rd_M = -1$
21: $freq_M = ratio_N$
22: **Return RP**

generation process, described in Algorithm1, ensures diverse and realistic memory access patterns for training.

The process of generating custom RPs consists of two main steps. The first step is to determine the locations of the Gaussian distributions (i.e., the means of the Gaussian distributions), and the second step is to generate Gaussian distributions based on the determined locations. This process is presented in **Algorithm 1**.

In Algorithm 1, lines 1–2 describe the process of generating RP data, illustrating the input and output. It shows that N and $MaxRD$ are used as input parameters to generate custom RP data. The format of the RP data consists of M RD values and their corresponding frequencies, as shown in the table in Fig. 1.

Lines 3–12 detail the first stage, where the positions of the Gaussian distributions are set. The number of position of the Gaussian distributions depends on the N parameter. In line 6, the RND_i function generates N values by producing integers uniformly distributed between 0 and $MaxRD$. These values are stored in pos_n. In line 7, the RND_f function generates floating-point value uniformly distributed between 0 and 1, determining the combination ratio $ratio_n$ of the Gaussian distributions. In other words, the ratio of the Gaussian distribution generated at position pos_n is defined as $ratio_n$. These two processes are repeated N times. In lines 10–11, the last RD is fixed at -1, and its ratio is determined in the same manner. The section where RD equals -1 must be included in the RP data. Finally, in line 12, the generated $N + 1$ ratio values are normalized so that their sum equals 1.

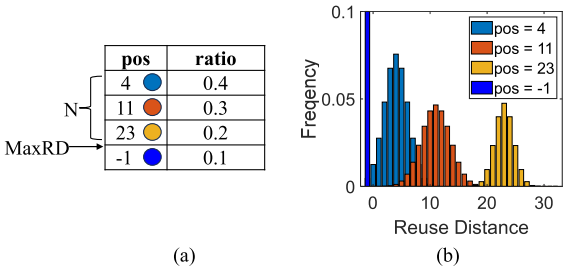

FIGURE 4. (a) Generate peak RD randomly when $N = 3$, $MaxRD = 32$ (b) Custom RP data.

Lines 13–21 describe the second stage, where Gaussian distributions are generated. Based on the positions and ratios set in the first stage, Gaussian distributions are generated. In line 15, M denotes the number of index points in the user-defined RP data. Thus, RP data generation is performed for RD values from 0 to $M - 1$. In line 17, Gaussian distributions are generated with the position values as the mean and a standard deviation. The function $gaussian(x, \mu, \sigma)$ returns the probability density value at x for a Gaussian distribution with mean μ and standard deviation σ. Here, σ is randomly set within the range (0, 2). Finally, in lines 20–21, the frequency corresponding to the RD value of -1 is set to maintain the $ratio_N$ value defined in the first stage. The sum of all $freq_m$ values is guaranteed to equal 1.

Fig. 4 provides another explanation with a simple example. Assuming $N = 3$ and $MaxRD = 32$, three numbers are randomly selected from the range [0, 32]. In Fig. 4(a), the randomly chosen location values are 4, 11, and 23. After generating and normalizing the distribution ratios corresponding to these values, the Gaussian distribution results based on this data are shown in Fig. 4(b). This graph illustrates three Gaussian distributions with RD locations at 4, 11, and 23. Each of the three Gaussian distributions occupies a proportion defined by the previously selected ratio values.

2) GENERATING SYNTHETIC TRACES

In this step, synthetic traces are generated based on the created RP data. This process is aimed at creating input for a trace-based cyclic-simulator and consists of two main parts. The first part involves generating memory indices to be accessed based on custom RP. The second part involves generating a kernel that visits the corresponding memory addresses to complete the synthetic trace. The first part is described in detail in **Algorithm 2**.

Lines 1–2 of Algorithm 2 define the inputs and outputs for generating memory indices. The inputs are L and **RP**. Here, L represents the length of the memory indices, and **RP** refers to the custom RP data generated in the previous step. The output is expressed as **idx**, a vector format of length L.

Lines 4–21 describe the process of returning memory indices. First, in line 4, a stack is initialized to store arbitrary

Algorithm 2 Generate Memory Indexes for Synthetic Traces

1: **Input:** L, **RP** $= \{(rd_i, freq_i)\ | i = 0, 1, \ldots, M\}$
2: **Output: idx** $= (idx_0, idx_1, \ldots, idx_{L-1})$
3: $l \leftarrow 0$
4: **stack** $= [\ ]$
5: **while** $l < L$ **do**
6: Draw rd from $(rd_0, rd_1, \ldots)$ for $(freq_0, freq_1, \ldots)$
7: **while** $size$(**stack**) $< rd$ **do**
8: $idx_l = RND_i(0, ADDRS)$
9: **stack**.$push(idx_l)$
10: Update **RP** for $rd_i = -1$
11: Append **idx** with idx_l
12: $l \leftarrow l + 1$
13: **end while**
14: $idx_l =$ **stack**$[-rd]$
15: **stack**.$erase(idx_l)$
16: **stack**.$push(idx_l)$
17: Update **RP** for $rd_i = rd$
18: Append **idx** with idx_l
19: $l \leftarrow l + 1$
20: **end while**
21: **return idx**

memory references for processing RD. Memory indices are selected based on the rd value of the custom RP data and the probability of the corresponding $freq$. Since the sum of all $freq_i$ values is 1, they can be considered as probabilities. In line 6, one of the rd_i values is selected based on $freq_i$ at each time step, which is defined as rd.

Depending on the state of the stack, the algorithm is divided into two parts. The first case occurs when the stack is empty or its depth is smaller than the selected rd, in which case the process from lines 8–12 is performed. In this case, in line 8, a memory reference idx_l is selected uniformly within the address space $ADDRS$. Then in line 9, the selected idx_l is pushed onto the stack. If $rd = -1$, the reference is considered as being accessed for the first time, and **RP** is updated. In line 10, **RP** is updated based on the rd value, adjusting the ratio of the remaining index length. Finally, in line 11, the selected idx_l is added to the index vector **idx**.

The second case occurs when the stack is sufficiently filled. In this case, the process from lines 14–17 is performed. If the stack's depth matches the selected rd, the idx_l value at the depth corresponding to rd is retrieved from the stack. Then, in lines 15–16, the selected reference is moved to the top of the stack, and in line 17, **RP** is updated. The selected idx_l is added to the index vector **idx**. During subsequent iterations, **idx** is updated, and in line 6, a new rd value is selected. This process is repeated L times to generate the final memory index vector **idx**.

The process of updating **RP** follows the equation below. This equation uses the trace length L as a parameter and normalizes the $freq$ value at $(l+1)$-th based on the $freq$ value at l-th to update it.

$$freq_{j,l+1} = \begin{cases} \text{if } rd_j = rd, \\ \quad \max\left(freq_{j,l} \times \dfrac{L-l}{L-l-1} - \dfrac{1}{L-l-1}, 0\right) \\ \text{otherwise}, \\ \quad freq_{j,l} \times \dfrac{L-l}{L-l-1} \end{cases}$$

Table 3 illustrates a simple example of generating a memory index vector **idx** using a predefined RP. The initial RP is given as **RP** $=$ {**rd**; **freq**} $=$ {{0, 1, 2, 3, −1}; {0.1, 0.2, 0.3, 0.3, 0.1}}. At each time step l, a reuse distance rd is selected based on the probability distribution defined by **freq**.

When the selected rd exceeds the current stack depth (e.g., at $l = 0$), a new memory reference index is randomly generated and added to both the trace vector and the stack. If the stack depth is sufficient (e.g., from $l = 3$ onward), a value is reused from the stack at the depth corresponding to the selected rd, and the stack is updated accordingly to reflect recent usage.

After each memory access, the corresponding frequency in the RP is updated using the recurrence relation described earlier, gradually adjusting the probability distribution as the trace progresses. This iterative process results in a memory index vector **idx** of length L, whose reuse behavior aligns with the statistical properties of the initial RP.

TABLE 3. Example of memory index generation for synthetic traces.

l		0	1	2	3	4	5	…
rd		2	-	-	-	1	0	…
idx		40	16	25	40	25	25	…
RP		update **RP**						
rd	**freq**	update **freq**						
0	0.1	0.101	0.102	0.103	0.104	0.105	0.096	…
1	0.2	0.202	0.204	0.206	0.208	0.200	0.202	…
2	0.3	0.303	0.306	0.309	0.302	0.305	0.309	…
3	0.3	0.303	0.306	0.309	0.313	0.316	0.319	…
-1	0.1	0.091	0.082	0.072	0.073	0.074	0.074	…

The second part of generating synthetic traces involves creating a kernel that iterates through memory indices to extract the synthetic trace. The index vector **idx** generated in the previous step represents a set of memory reference indices based on custom RP data. Once the kernel iterating through memory addresses is executed, these can be converted to the input format required by the trace-based cycle simulator.

C. TRAINING STAGE

The second stage of MLCRP involves training a machine learning model. This step is described in the second row of Fig. 3. Here, the MLmodel is a machine learning-based model designed to predict cache performance. During the training phase, the MLmodel learns from the data and is then utilized in the inference phase to predict actual cache performance. Defining the inputs and outputs of the MLmodel and selecting

an appropriate machine learning model are crucial factors in determining prediction accuracy.

1) INPUT/OUTPUT DEFINITION

Clear definitions of inputs and outputs are essential to train the MLmodel effectively. As illustrated in Fig. 3, the inputs include custom RP information and cache configuration details. Custom RP information consists of parameters used in the generation process, such as N, $MaxRD$, and user-defined RPs. Cache configuration details provide specifics about the set, way, and line size of the cache. The input data is represented in vector form, including normalized values of N and $MaxRD$, as well as the $freq_i$ generated by **Algorithm 1** from the RP vector. Cache configuration details are also normalized to values between 0 and 1.

The output of the MLmodel is labeled with cache performance metrics derived from the input RP information. These labels are based on cache performance values obtained by analyzing synthetic traces generated from the custom RP using a cycle-accurate simulator. The outputs include the cache miss rate and MSHR merge rate, provided in a (2, 1) vector format.

2) MLmodel SELECTION

For cache performance modeling, a machine learning model capable of accurately capturing the nonlinear relationship between inputs and outputs is required. Multi-Layer Perceptron (MLP) is well-suited for nonlinear problems due to its use of nonlinear activation functions and multi-layer structures that excel in learning high-dimensional patterns. Traditional machine learning methods such as SVR (Support Vector Regression), RF (Random Forest), and GB (Gradient Boosting) each address nonlinear data in distinct ways. SVR leverages kernel methods to map nonlinear data into a higher-dimensional space, making it particularly effective for high-dimensional data. RF and GB models utilize ensembles of decision trees to learn various data patterns and effectively handle nonlinear relationships through their tree-based structures.

These models are strong candidates for the MLmodel as they offer high prediction performance on nonlinear data. The performance and applicability of each model are discussed in detail in section V, where experimental results are used for comparative evaluation.

D. INFERENCE STAGE

The inference stage is the final step of MLCRP and utilizes the trained MLmodel to predict cache performance for real applications. As illustrated in Fig. 3, the model transitions from training to a learned state and receives input data extracted from actual workloads. The inference stage plays a crucial role in validating the generalization capability of MLCRP across unseen cache configurations and diverse workloads.

1) INPUT FEATURE EXTRACTION FROM TARGET APPLICATIONS

To enable practical usage, a preprocessing step extracts RP data from the target application trace. This includes computing the number of distinguishable RD clusters N, the maximum RD value $MaxRD$, and the normalized frequency vector $(freq_0, freq_1, \ldots)$. These values are extracted by dividing the RD trace into fixed-length intervals and analyzing each segment independently. This approach captures temporal variations in memory access behavior more effectively than static RP extraction.

By tuning the interval length, the temporal resolution of performance prediction can be controlled, allowing MLCRP to track dynamic cache behavior at a finer granularity if necessary. Distinct peaks in the RD histogram are identified using clustering techniques or peak detection, and their count determines the value of N.

TABLE 4. GPU applications from Rodinia 3.1*, Polybench† and Tango‡.

Application	Abbr.	Application Trace Size	MLmodel Input Size
AlexNet‡	*AN*	22GB	6.2MB
atax†	*ATAX*	213MB	5.19KB
bfs*	*BFS*	1.7GB	263.87KB
bicg†	*BICG*	214MB	5.19KB
backprop*	*BP*	229MB	70.13KB
CifarNet‡	*CN*	489MB	157.71KB
dwt2d*	*DWT*	242MB	134.65KB
gaussian*	*GAUSS*	438MB	4.59MB
gesummv†	*GES*	287MB	4.99KB
lud*	*LUD*	390MB	7.5KB
mvt†	*MVT*	213MB	5.19KB
nw*	*NW*	394MB	32.72KB
particlefilter_float*	*PFF*	329MB	907.14KB
particlefilter_naive*	*PFN*	45MB	12.08KB
ResNet‡	*RN*	129GB	16.71MB
SqueezeNet‡	*SQN*	24GB	8.18MB
srad*	*SRAD*	14GB	17.17MB

TABLE 5. Simulation cache configurations.

Cache Size	1, 2, …, 128 KB per SM
Cache Associativity	1,2,…, 1024 ways

The final input vector for the MLmodel consists of normalized RP features $(N, MaxRD, freq_i)$, along with normalized cache configuration parameters.

2) ANALYSIS OF PRE-PROCESSING OVERHEAD

The pre-processing complexity is manageable and split into two parts. First, RP extraction is performed alongside trace collection, incurring no additional cost. Second, identifying RD clusters and constructing the input vector scales with the length of $freq_i$, which itself is independent of trace length.

As a result, the end-to-end preprocessing pipeline remains efficient.

Unlike traditional simulators, which require extensive trace replay and full-cycle evaluation, MLCRP enables near-instantaneous performance prediction once RP data is available. This property makes it highly suitable for rapid DSE or real-time cache tuning in GPU systems.

Since MLCRP is trained on synthetic traces with diverse RP patterns and cache configurations, it generalizes well to unseen application behaviors and hardware settings. This ensures practical applicability even in scenarios where the exact workload or architecture was not encountered during training.

V. EVALUATION AND EXPERIMENTAL RESULTS

We evaluate MLCRP using a diverse set of GPU workloads to demonstrate its practical applicability and scalability in high-performance memory systems.

(*N*,*MaxRD*) : Each cell contains training data

MaxRD \ N	2	3	4	5	6	7	8	9	10	11	12
4096	(2,4096)	(3,4096)	(4,4096)	(5,4096)	(6,4096)	(7,4096)	(8,4096)	(9,4096)	(10,4096)	(11,4096)	(12,4096)
1024	(2,1024)	(3,1024)	(4,1024)	(5,1024)	(6,1024)	(7,1024)	(8,1024)	(9,1024)	(10,1024)	(11,1024)	(12,1024)
128	(2,128)	(3,128)	(4,128)	(5,128)	(6,128)	(7,128)	(8,128)	(9,128)	(10,128)	(11,128)	(12,128)
16	(2,16)	(3,16)	(4,16)	(5,16)	(6,16)	(7,16)	(8,16)	(9,16)	(10,16)	(11,16)	(12,16)

FIGURE 5. (*N*, *MaxRD*) combinations used for data preparation.

A. EXPERIMENTAL SETUP

1) SIMULATION ENVIRONMENT

We validate the proposed MLCRP framework using a diverse set of GPU applications. A total of 17 GPU applications are used, including 9 applications from Rodinia 3.1 [37], which consists of various kernels focused on scientific computation, 4 applications from Polybench [38], which primarily deals with linear algebra, and 4 applications from the DNN benchmark suite Tango [39], which includes deep neural network workloads. By employing this broad range of GPU applications, we aim to demonstrate the generalizability of the proposed framework. The list of applications used in the experiments, along with their abbreviations and sizes, is provided in Table 4.

Our experiments are conducted on a desktop equipped with an Intel(R) Core(TM) i9-9900K CPU running at 3.6GHz, and the operating system is Linux Ubuntu 18.04 LTS. To extract memory traces of actual applications, we utilize the NVBit program [40], and for profiling, we employ Accel-Sim [5], a trace-driven simulator based on GPGPU-Sim v4.0. The memory trace extraction is performed once using a GeForce RTX 3090 with CUDA v11.4.

Additionally, a cycle-accurate simulator, Accel-Sim, was used to perform various measurements under different simulation configurations, and these results were compared with MLCRP outcomes to evaluate cache performance. Specifically, the simulator was configured as shown in Table 5, with the cache size per SM varying from 1 KB to 128 KB and cache associativity adjusted from 1-way to 1024-way. Meanwhile, the number of SMs was fixed at 82, based on modern GPU architectures, to maintain consistency across experiments.

2) DATA PREPARATION

In MLCRP, the input parameters N and $MaxRD$, which are required for training data generation, are utilized in the data preparation process to reflect various application environments. Fig. 5 illustrates the (N, $MaxRD$) combinations used to ensure data diversity. The horizontal axis represents N, while the vertical axis represents $MaxRD$, with the generated data spanning the range of (N, $MaxRD$) values from (1,12) to (1,4096). The RP generation method utilizing these (N, $MaxRD$) combinations is described in Algorithm 1.

The trace length L for the training data is fixed at 10,000. Additionally, for comparative analysis, each combination in Fig. 5 is considered as a single cell, and the number of training data samples per cell is adjusted to 5, 10, 20, and 50 to analyze the results. This approach helps in exploring the optimal training environment based on parameter density and the number of training samples.

3) TRAINING AND INFERENCE

Four machine learning models for MLmodel are employed: traditional machine learning models such as SVM, RF, and GB, as well as a neural network-based model, MLP. The hyperparameters for each model are based on standard configurations from prior research in system performance prediction. The hyperparameters for each model are as follows:

- **MLP** The model uses 3 hidden layers with 1024, 128, and 32 nodes, respectively. A dropout rate of 0.5 is applied, and ReLU is used as the activation function. Since the output layer predicts values between 0 and 1, a sigmoid function is applied before the output layer. The learning rate is set to 0.001, and the batch size is 256. 80% of the total data is used for training, while the remaining 20% is set as the test set to verify convergence.
- **SVM** A radial basis function kernel is used, with a no-penalty parameter of 0.1 and a penalty parameter of 1.0.
- **RF** The number of trees used is 100, with no limit on the tree depth. The minimum number of samples required to split an internal node is 2.
- **GB** The number of boosting stages is set to 100, with a maximum depth of 3 for each decision tree. The minimum number of samples required to split an internal node is 2, and the learning rate is 0.1.

The performance of the MLmodel is compared with various cache models. The comparison includes PPT-GPU-Mem [15], a mathematical analytical model, and Accel-Sim, a cycle-accurate simulator. The prediction results and simulation times of these models are evaluated against those of the MLmodel in MLCRP. While PPT-GPU-Mem is limited to predicting only cache miss rates, Accel-Sim is capable

of predicting both cache miss rates and MSHR merge rates, making it suitable for a direct performance comparison with the MLmodel.

Performance metrics are compared using mean absolute error (MAE) and R^2 score (coefficient of determination). MAE is the mean of the absolute differences between the predicted and actual values, evaluating the absolute prediction error of the model. MAE is defined as follows:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

where n is the number of data points, y_i represents the actual values, and $\hat{y}_i$ represents the predicted values.

On the other hand, the R^2 score measures how well the model's predictions explain the variance in the actual values. An R^2 score closer to 1 indicates a better performing model. R^2 is defined as:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2}$$

where $\bar{y}$ is the mean of the actual values.

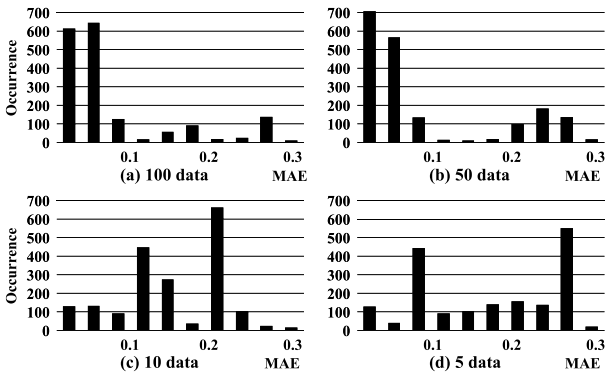

FIGURE 6. Error occurrence depending on training data size: (a) 100, (b) 50, (c) 10, (d) 5.

B. MLCRP PARAMETER SENSITIVITY

1) DATA PREPARATION STAGE

The performance of machine learning models is significantly influenced by the amount of generated training data. Fig. 6 presents the results obtained when generating 5, 10, 50, and 100 training data samples per (N, $MaxRD$) parameter combination.

When the unit data size is 100 (Fig. 6(a)), most errors occur below 0.6. When the data size is reduced to 50 (Fig. 6(b)), a similar trend is observed, but the error occurrence frequency slightly increases in the 0.2–0.3 range. As the data size decreases to 10 (Fig. 6(c)) and 5 (Fig. 6(d)), errors exceeding 0.2 occur more frequently, and the performance degradation becomes more pronounced.

Fig. 7(a) shows the MAE results based on cache configurations. When there are five training data samples per cell, the MAE is approximately 1.6%, and it gradually decreases as the number of training samples increases. When using 50 and 100 samples per cell, the MAE remains nearly identical; however, using 100 samples doubles the time complexity. Therefore, despite a slight loss in accuracy, using 50 samples per cell may be a more efficient choice.

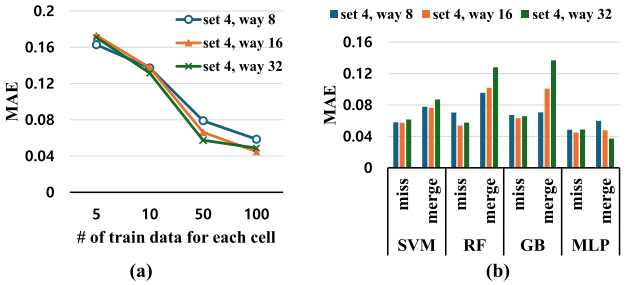

FIGURE 7. MAE variation based on (a) the training data size and (b) different machine learning models.

An analysis of error distribution based on dataset size reveals that as the dataset size decreases, errors tend to be more evenly distributed, with errors exceeding 0.2 occurring at the highest frequency (Fig. 6(c), (d)). In contrast, when the dataset size increases beyond 50, most errors remain below 0.1, and the distribution variance significantly decreases. This indicates that greater diversity in training data positively impacts model prediction performance. For instance, when the dataset size is 5, the average error is 0.35 with a standard deviation of 0.12, whereas with a dataset size of 100, the average error decreases to 0.08 with a standard deviation of 0.05.

An analysis of the relationship between training dataset size and prediction accuracy shows that as the dataset size increases, MAE tends to decrease. However, after exceeding a dataset size of 50, a saturation effect is observed, where the rate of MAE reduction diminishes significantly (Fig. 7(a)). For example, increasing the dataset size from 10 to 50 reduces MAE by approximately 1.2%, but increasing it from 50 to 100 results in only a 0.1% additional reduction. This suggests that generating additional samples beyond a dataset size of 50 contributes minimally to accuracy improvement. Therefore, considering both performance and efficiency, setting the dataset size to 50 may be the most appropriate choice.

2) TRAINING MLmodel

Fig. 7(b) illustrates the results obtained using traditional machine learning models such as SVM, RF, and GB. All three models demonstrated relatively good performance, with MAE within 10%. However, MLP consistently achieved the lowest MAE across all experimental configurations. These results suggest that MLP effectively captures the nonlinear relationships between RP data and cache performance. Notably, MLP exhibited more stable predictive performance than the other models, even in cache configurations with high complexity.

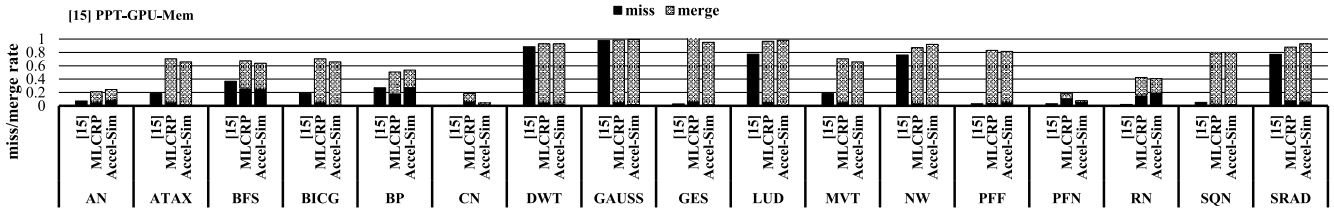

FIGURE 8. When L1 cache size is 8KB per SM, associativity is 8 and line size is 128B, miss/merge rate prediction.

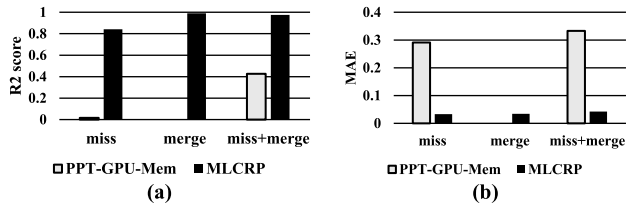

FIGURE 9. Performance comparison with Accel-Sim based on (a) R^2 score and (b) MAE.

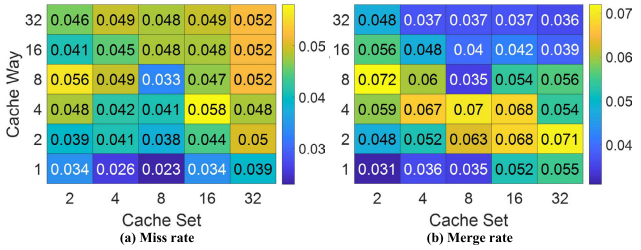

FIGURE 10. MAE of MLCRP across all applications under different cache configurations: (a) miss rate, (b) merge rate.

Based on these findings, subsequent experiments will set the unit training data size to 50. Additionally, MLP will be adopted as the MLmodel to validate its performance across various cache configurations. This decision reflects the advantage of MLP's nonlinear learning capability in predicting performance.

C. PERFORMANCE OF MLCRP

Fig. 8 presents the performance prediction results for a cache configuration with 8 sets, 8 ways, and an 8 KB size. The prediction results for each application are compared with those from the analytical model PPT-GPU-Mem [15] and the cycle-accurate simulator Accel-Sim.

The proposed MLCRP model can predict both cache miss rates and MSHR merge rates, demonstrating high accuracy across various applications. In contrast, PPT-GPU-Mem is limited to predicting only the miss rate, providing a relatively restricted analysis. Fig. 8 shows that MLCRP provides predictions closer to Accel-Sim results compared to PPT-GPU-Mem.

A quantitative analysis of the relative error between MLCRP and Accel-Sim revealed that for most applications, MLCRP maintained an error rate of less than 5% compared to Accel-Sim. For simpler patterns such as BFS, the error was as low as 1%. This result indicates that MLCRP effectively learns the complex memory access characteristics associated with different cache configurations. Fig. 9 provides a quantitative comparison of the prediction results, where Figs. 9(a) and 9(b) depict the R^2 score and MAE, respectively, comparing MLCRP with PPT-GPU-Mem based on the data from Fig. 8. In all cases, MLCRP achieved higher R^2 scores and lower MAE values than PPT-GPU-Mem.

Notably, MLCRP recorded an R^2 score of 0.975 for both miss rate and merge rate predictions, significantly outperforming PPT-GPU-Mem, which had an R^2 score of 0.426. Additionally, MLCRP achieved an MAE of 0.043, which is lower than PPT-GPU-Mem's MAE of 0.333, confirming MLCRP's superior prediction accuracy over existing analytical models.

The overall performance of the cache configuration model is presented in Fig. 10. Figs. 10(a) and 10(b) illustrate the MAE for miss rate and merge rate predictions across all applications as the number of cache sets and ways varies. These results demonstrate that MLCRP maintains a high prediction accuracy within a 5% error margin compared to Accel-Sim. In particular, the stable MAE for miss rate and merge rate predictions, even as the set and way sizes increase, highlights MLCRP's scalability and reliability.

Finally, the ability of MLCRP to simultaneously predict both miss rate and merge rate while maintaining low errors further underscores its practical applicability.

D. SIMULATION TIME AND TRAINING COST ANALYSIS

Fig. 11 presents a log-scale comparison of simulation times for PPT-GPU-Mem, MLCRP, and Accel-Sim. Since all three models include a trace extraction process, the measured simulation time reflects only the stages following trace extraction. While Accel-Sim's simulation time increases exponentially with application trace size, PPT-GPU-Mem employs a simple formula, resulting in extremely short simulation times. MLCRP, although slightly slower than PPT-GPU-Mem, achieves significantly faster simulation speeds compared to Accel-Sim.

Notably, for applications with simple memory access patterns such as BFS, there was almost no performance difference between MLCRP and PPT-GPU-Mem. In contrast, for applications with more complex cache access patterns,

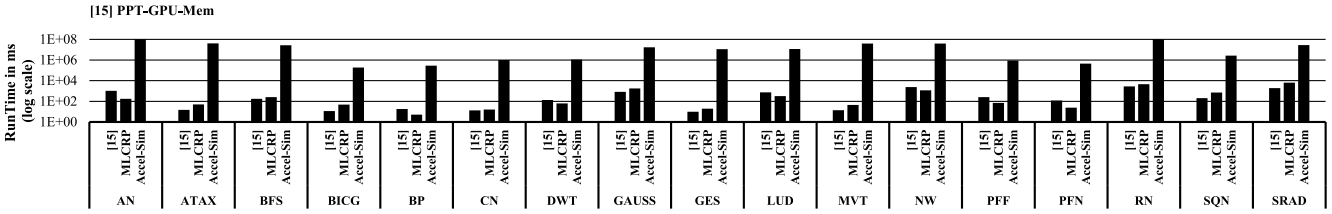

FIGURE 11. When L1 cache size is 8KB per SM, associativity is 8 and line size is 128B, simulation time in mili-seconds.

such as GES and LUD, MLCRP demonstrated higher accuracy than PPT-GPU-Mem. This result indicates that MLCRP effectively captures complex data characteristics.

MLCRP leverages GPU acceleration, achieving a speedup of over five orders of magnitude faster compared to Accel-Sim for most applications, demonstrating its high efficiency. As memory trace size increases, the speed improvement becomes even more significant compared to CPU-based execution. These results suggest that MLCRP is a suitable solution for large-scale data processing and complex design space exploration simulations. Furthermore, MLCRP utilizes GPU acceleration and parallel processing to significantly reduce simulation and training costs, providing a crucial advantage in high-performance computing environments.

From a training cost perspective, MLCRP's training dataset includes trace data of length 10,000 (size 73MB), with each individual dataset labeling requiring an average of 627.17 seconds (approximately 10.45 minutes). When generating 100 data samples using 10 cores in parallel, the total processing time is approximately 100 minutes. In comparison, simulating AlexNet using Accel-Sim requires approximately 189,348 seconds (about 52 hours), confirming that MLCRP offers significantly higher efficiency.

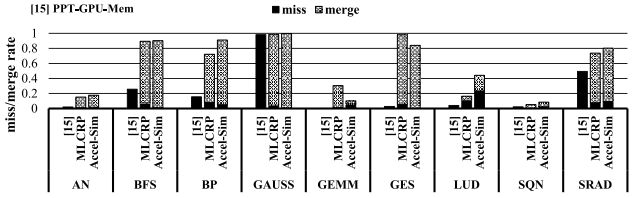

FIGURE 12. When L2 cache is configured with 16 sets, 64-way associativity, and 128B line size (totaling 128KB L2 cache), miss/merge rate prediction results for selected GPU applications.

E. EXTENSION TO L2 CACHE MODELING

To assess the extensibility of MLCRP beyond private L1 caches, we conducted a proof-of-concept experiment on shared L2 cache modeling. As shown in Fig.12, MLCRP was directly applied to the L2 memory traces of selected GPU workloads without modification.

RP were computed at fixed intervals from L2 traces, following the same segmentation method used for L1. Unlike L1, however, L2 traces reflect more complex and non-stationary behaviors due to shared access across SMs and upstream effects from L1 cache dynamics.

Despite these challenges, MLCRP achieved reasonable prediction accuracy across diverse applications. Some degradation compared to L1 was observed, which we attribute to the lack of detailed L1 access context and unmodeled inter-cache interactions. Nonetheless, the results validate the model's potential to generalize across cache hierarchies with minimal changes.

TABLE 6. The miss rate ranking of *GES*.

(set, way)	Accel-Sim	MLCRP	PPT-GPU-Mem
(2, 512)	R1	R1	R1
(1, 1024)	R2	R2	R2
(1, 512)	R3	R3	R9
(4, 256)	R4	R5	R3
(2, 256)	R5	R7	R7
(8, 128)	R6	R4	R4
(4, 128)	R7	R6	R8
(1, 128)	R8	R8	R10
(4, 64)	R9	R9	R6
(2, 64)	R10	R10	R5
ρ[1]	-	0.939	0.491

[1] Spearman's Rank Correlation Coefficient

F. RANK ACCURACY ANALYSIS

MLCRP can also be applied to cache optimization design problems. For instance, when evaluating 10 different cache configurations, MLCRP can quickly determine their rankings. Table 6 presents the miss rate rankings for the GES application, comparing the rank correlation coefficient measured against Accel-Sim. MLCRP achieved a remarkably high correlation coefficient of 0.939, whereas PPT-GPU-Mem recorded 0.491, showing no clear trend.

Notably, MLCRP accurately predicts rankings even for complex cache configurations, significantly improving the efficiency of design space exploration. When comparing the top three and bottom three ranked configurations, MLCRP's predicted rankings aligned with those of Accel-Sim. This result indicates that MLCRP can reliably determine design priorities.

Some minor discrepancies were observed in the middle-ranked configurations, likely due to differences in miss rate predictions for specific cache configurations. However,

these errors have minimal impact on the overall design space exploration and do not significantly affect MLCRP's efficiency and accuracy.

These findings demonstrate that MLCRP can be effectively utilized for cache design space exploration. The high rank correlation coefficient and precise performance differentiation suggest that MLCRP has strong potential as a key tool for design optimization problems.

VI. CONCLUSION

This paper proposes MLCRP, a machine learning-based cache performance modeling framework that bridges the gap between analytic and simulation-based methods. MLCRP utilizes RP features to generate training data and enables accurate prediction of key cache metrics—including miss rates and MSHR merge rates—across a wide range of configurations.

Experimental evaluations demonstrate that MLCRP achieves prediction MAE below 5%, an R^2 score of 0.975, and over 10,000× speedup compared to Accel-Sim, while also enabling effective cache configuration ranking for design space exploration. These results confirm that MLCRP can serve as a scalable and high-fidelity alternative to traditional simulators.

With its ability to capture non-stationary memory behaviors, support diverse cache configurations, and deliver accurate predictions with low overhead, MLCRP provides a practical and efficient solution for cache modeling in high-performance GPU systems. It offers significant advantages in simulation time, generalization, and applicability, making it a valuable tool for large-scale design space exploration and optimization.

REFERENCES

[1] R. Ausavarungnirun, J. Landgraf, V. Miller, S. Ghose, J. Gandhi, C. J. Rossbach, and O. Mutlu, "Mosaic: A GPU memory manager with application-transparent support for multiple page sizes," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*. New York, NY, USA: ACM, Oct. 2017, pp. 136–150.

[2] M. Khairy, M. Zahran, and A. Wassal, "SACAT: Streaming-aware conflict-avoiding thrashing-resistant GPGPU cache management scheme," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 6, pp. 1740–1753, Jun. 2017.

[3] J. Lowe-Power et al., "The gem5 simulator: Version 20.0+," 2020, *arXiv:2007.03152*.

[4] M. A. Raihan, N. Goli, and T. M. Aamodt, "Modeling deep learning accelerator enabled GPUs," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2019, pp. 79–92.

[5] M. Khairy, Z. Shen, T. M. Aamodt, and T. G. Rogers, "Accel-sim: An extensible simulation framework for validated GPU modeling," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit. (ISCA)*, May 2020, pp. 473–486.

[6] M. Brehob and R. Enbody, "An analytical model of locality and caching," Dept. Comput. Sci. Eng., Michigan State Univ., East Lansing, MI, USA, Tech. Rep. MSU-CSE-99-31, 1999.

[7] O. Navarro, J. Yudi, J. Hoffmann, H. G. M. Hernandez, and M. Hübner, "A machine learning methodology for cache memory design based on dynamic instructions," *ACM Trans. Embedded Comput. Syst.*, vol. 19, no. 2, pp. 1–20, Mar. 2020.

[8] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, May 2003, pp. 245–257.

[9] C. CaBcaval and D. A. Padua, "Estimating cache misses and locality using stack distances," in *Proc. 17th Annu. Int. Conf. Supercomputing*, Jun. 2003, pp. 150–159.

[10] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proc. 36th Annu. Int. Symp. Comput. Archit.*, Jun. 2009, pp. 152–163.

[11] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-M.-W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *Proc. 15th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Jan. 2010, pp. 105–114.

[12] J. Lee, Y. Ha, S. Lee, J. Woo, J. Lee, H. Jang, and Y. Kim, "Gcom: A detailed GPU core model for accurate analytical modeling of modern GPUS," in *Proc. 49th Annu. Int. Symp. Comput. Archit.*, Aug. 2022, pp. 424–436.

[13] Y. Arafa, A.-H. Badawy, A. ElWazir, A. Barai, A. Eker, G. Chennupati, N. Santhi, and S. Eidenbenz, "Hybrid, scalable, trace-driven performance modeling of GPGPUs," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal. (SC21)*, Nov. 2021, pp. 1–15.

[14] T. Tang, X. Yang, and Y. Lin, "Cache miss analysis for GPU programs based on stack distance profile," in *Proc. 31st Int. Conf. Distrib. Comput. Syst.*, Jun. 2011, pp. 623–634.

[15] Y. Arafa, A.-H. Badawy, G. Chennupati, A. Barai, N. Santhi, and S. Eidenbenz, "Fast, accurate, and scalable memory modeling of GPGPUs using reuse profiles," in *Proc. 34th ACM Int. Conf. Supercomputing*, Jun. 2020, pp. 1–12.

[16] A. Ghosh and T. Givargis, "Analytical design space exploration of caches for embedded systems," in *Proc. Design*, Dec. 2003, pp. 650–655.

[17] A. Janapsatya, A. Ignjatovic, and S. Parameswaran, "Finding optimal L_1 cache configuration for embedded systems," in *Proc. Asia South Pacific Conf. Design Autom.*, Jul. 2006, pp. 796–801.

[18] M. S. Haque, A. Janapsatya, and S. Parameswaran, "SuSeSim: A fast simulation strategy to find optimal L_1 cache configuration for embedded systems," in *Proc. 7th IEEE/ACM Int. Conf. Hardw./Softw. Codesign Syst. Synth.*, Oct. 2009, pp. 295–304.

[19] X. Li, H. S. Negi, T. Mitra, and A. Roychoudhury, "Design space exploration of caches using compressed traces," in *Proc. 18th Annu. Int. Conf. Supercomputing*, Jun. 2004, pp. 116–125.

[20] C. Nugteren, G.-J. van den Braak, H. Corporaal, and H. E. Bal, "A detailed GPU cache model based on reuse distance theory," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2014, pp. 37–48.

[21] M. Kiani and A. Rajabzadeh, "Efficient cache performance modeling in GPUs using reuse distance analysis," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 4, pp. 1–24, Dec. 2018.

[22] K. Beyls and E. D'Hollander, "Reuse distance as a metric for cache behavior," in *Proc. IASTED Conf. Parallel Distrib. Comput. Syst.*, Jan. 2001, pp. 350–360.

[23] E. Ïpek, S. A. McKee, R. Caruana, B. R. de Supinski, and M. Schulz, "Efficiently exploring architectural design spaces via predictive modeling," *ACM SIGOPS Operating Syst. Rev.*, vol. 40, no. 5, pp. 195–206, 2006.

[24] D. Li, S. Yao, Y.-H. Liu, S. Wang, and X.-H. Sun, "Efficient design space exploration via statistical sampling and AdaBoost learning," in *Proc. 53rd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2016, pp. 1–6.

[25] C. Bai, Q. Sun, J. Zhai, Y. Ma, B. Yu, and M. D. F. Wong, "BOOM-explorer: RISC-V BOOM microarchitecture design space exploration framework," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Design (ICCAD)*, Nov. 2021, pp. 1–9.

[26] Q. Niu, J. Dinan, Q. Lu, and P. Sadayappan, "PARDA: A fast parallel reuse distance analysis algorithm," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, May 2012, pp. 1284–1294.

[27] S. Ghosh, M. Martonosi, and S. Malik, "Cache miss equations: A compiler framework for analyzing and tuning memory behavior," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 4, pp. 703–746, Jul. 1999.

[28] J. J. Pieper, A. Mellan, J. M. Paul, D. E. Thomas, and F. Karim, "High level cache simulation for heterogeneous multiprocessors," in *Proc. 41st Annu. Design Autom. Conf.*, Jun. 2004, pp. 287–292.

[29] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A performance analysis framework for identifying potential benefits in GPGPU applications," in *Proc. 17th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, Feb. 2012, pp. 11–22.

[30] S. S. Baghsorkhi, M. Delahaye, W. D. Gropp, and H. Wen-mei, "Analytical performance prediction for evaluation and tuning of gpgpu applications," in *Proc. Workshop Exploiting Parallelism GPUs Hardw.-Assist. Methods (EPHAM), Conjunct Int. Symp. Code Gener. Optim. (CGO)*, Jun. 2009.

[31] C. Ding and T. Chilimbi, "A composable model for analyzing locality of multi-threaded programs," Microsoft Res., Tech. Rep. MSR-TR-2009-107, 2009.
[32] Y. Jiang, E. Z. Zhang, K. Tian, and X. Shen, "Is reuse distance applicable to data locality analysis on chip multiprocessors?" in *Proc. 19th Int. Conf. Joint Compiler Construct. Eur. Conf. Theory Pract. Softw.*, Paphos, Cyprus. Cham, Switzerland: Springer, Jan. 2010, pp. 264–282.
[33] M.-J. Wu and D. Yeung, "Identifying optimal multicore cache hierarchies for loop-based parallel programs via reuse distance analysis," in *Proc. ACM SIGPLAN Workshop Memory Syst. Perform. Correctness*, Jun. 2012, pp. 2–11.
[34] M.-J. Wu, M. Zhao, and D. Yeung, "Studying multicore processor scaling via reuse distance analysis," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 499–510, Jun. 2013.
[35] J. Lew, D. Shah, S. Pati, S. Cattell, M. Zhang, A. Sandhupatla, C. Ng, N. Goli, M. D. Sinclair, T. G. Rogers, and T. Aamodt, "Analyzing machine learning workloads using a detailed GPU simulator," 2018, *arXiv:1811.08933*.
[36] D. Wang and W. Xiao, "A reuse distance based performance analysis on GPU L_1 data cache," in *Proc. IEEE 35th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Dec. 2016, pp. 1–8.
[37] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2009, pp. 44–54.
[38] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Proc. Innov. Parallel Comput. (InPar)*, May 2012, pp. 1–10.
[39] A. Karki, C. P. Keshava, S. M. Shivakumar, J. Skow, G. M. Hegde, and H. Jeon, "Detailed characterization of deep neural networks on GPUs and FPGAs," in *Proc. 12th Workshop Gen. Purpose Process. Using GPUs*, Apr. 2019, pp. 12–21.
[40] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A dynamic binary instrumentation framework for NVIDIA GPUs," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 372–383.

MINJUNG CHO received the B.S. degree from Yonsei University, Seoul, South Korea, in 2017, where she is currently pursuing the Ph.D. degree in electrical and electronic engineering. Her current research interests include memory architecture and design.

EUI-YOUNG CHUNG (Member, IEEE) received the B.S. and M.S. degrees in electronics and computer engineering from Korea University, Seoul, South Korea, in 1988 and 1990, respectively, and the Ph.D. degree in electrical engineering from Stanford University, Stanford, CA, in 2002.

From 1990 to 2005, he was a Principal Engineer with the SoC Research and Development Center, Samsung Electronics, Yongin, South Korea. He is currently a Professor with the School of Electrical and Electronics Engineering, Yonsei University, Seoul. His research interests include system architecture, bio-computing, and VLSI design, including all aspects of computer-aided design with the special emphasis on low-power applications and flash memory applications.