# Offloading Embedding Lookups to Processing-In-Memory for Deep Learning Recommender Models

by

Niloofar Zarif

BSc. of Computer Engineering, Sharif University of Technology, 2019

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

**Master of Applied Science**

in

THE FACULTY OF APPLIED SCIENCE
(Electrical and Computer Engineering)

The University of British Columbia
(Vancouver)

August 2023

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

**Offloading Embedding Lookups to Processing-In-Memory for Deep Learning Recommender Models**

submitted by **Niloofar Zarif** in partial fulfillment of the requirements for the degree of **Master of Applied Science** in **Electrical and Computer Engineering**.

**Examining Committee:**

Alexandra Fedorova, Professor, Electrical and Computer Engineering, UBC
*Supervisor*

Prashant Nair, Assistant Professor, Electrical and Computer Engineering, UBC
*Supervisory Committee Member*

Sathish Gopalakrishnan, Associate Professor, Electrical and Computer Engineering, UBC
*Supervisory Committee Member*

# Abstract

Recommender systems are an essential part of many industries and businesses. Generating accurate recommendations is critical for user engagement and business revenue. Currently, deep learning recommender models are commonly used, but they face challenges in processing and representing categorical data, which is a significant portion of the data used by these models. *Embedding layers* are often used to handle these complications by storing the numerical representation of different categories of a feature in a reduced vector space. Vectors representing all the categories of a feature in the reduced vector space will be stored in a tabular structure named *embedding table*. The operation of fetching the vector representation of a category from the embedding table and pooling them is called *embedding lookup*. However, embedding lookups have large memory footprints and require high memory bandwidth, leading to high latency and low throughput.

We have developed a new system called ***PIM-Rec*** to address these challenges by using the first commercially available Processing-In-Memory (*PIM*) capable DRAM modules for embedding lookups. *PIM-Rec* is the first system to use such DRAM modules, and it has shown an 80% decrease in end-to-end inference cycle latency and an 80% increase in latency-bound throughput compared to the standard CPU-only implementation. PIM DRAM modules are a good candidate for handling embedding lookups, especially with the recent drastic size increase of embedding tables. Although *PIM-Rec* faced obstacles, it offers a realistic solution and analysis while discovering the obstacles and projecting them. This new system provides a promising solution for improving the efficiency of recommender systems and reducing the load they incur in data centers.

# Lay Summary

Personalized recommender systems are present in many platforms in different shapes and consume a considerable portion of AI cycles within datacenters. Most recommender systems use *deep learning models* to generate personalized recommendations. The deep learning models used for personalized recommendation are commonly memory-intensive. A part of such models called the *embedding layer* is mostly responsible for making them memory-intensive. Up to 80% of a model's inference cycle latency can be taken up by the embedding layer operations. Bringing computation closer to memory helps with reducing latency by defying the memory-wall. **PIM-Rec** uses the first commercially available dynamic random access memory (DRAM) modules that are capable of simple data processing in memory to accelerate inference on embedding layers. A feature of modules used by PIM-Rec is the scaling of compute power with memory size. This translates into the ability to keep up with the rapid pace at which embedding layer sizes are growing. PIM-Rec can reduce the inference cycle latency by 89% and increase the throughput by 9.61X relative to the baseline CPU implementation.

# Preface

This thesis represents an original, unpublished, and independent work by the author, N. Zarif. I would like to acknowledge the invaluable contributions of my collaborators, Justin Wong and Dr. Alexandra Fedorova.

Dr. Alexandra Fedorova has served as my supervisor, providing guidance and expertise throughout the research process. Her profound knowledge of computer systems has been instrumental in shaping the direction of this study, ensuring its rigor and integrity. I am deeply grateful for her insightful feedback and continuous support.

I would also like to express my appreciation to Justin Wong for his remarkable assistance in implementing our experimental framework and conducting experiments. His technical expertise and meticulous attention to detail have greatly contributed to the successful completion of this research.

# Table of Contents

# List of Tables

# List of Figures

# Glossary

This glossary uses the handy `acroynym` package to automatically maintain the glossary. It uses the package's `printonlyused` option to include only those acronyms explicitly referenced in the LaTeX source. To change how the acronyms are rendered, change the `\acsfont` definition in `diss.tex`.

**AI** Artificial Intelligence

**API** Application Programming Interface

**ASIC** Application Specific Integrated Circuit

**BSC** Bachelor of Science

**CPU** Central Processing Unit

**DIMM** Dual In-Line memory module

**DLRM** Deep Learning Recommender Model

**DMA** Direct Memory Access

**DPU** Data Processing Unit

**DRAM** Dynamic Random Access Memory

**FIM** Function In Memory

**FLOPS** Floating Points Per Second

**FPGA** Field Programmable Gate Array

| | |
|---|---|
| **GPU** | Graphics Processing Unit |
| **HBM** | High Bandwidth Memory |
| **IPC** | Instructions Per Cycle |
| **L1D** | Level 1 Data (cache) |
| **LLC** | Last Level Cache |
| **ML** | Machine Learning |
| **MLP** | Multi-Layer Perceptron |
| **MPKI** | Misses per 1000 Instructions |
| **MRAM** | Main Random Access Memory |
| **MTIA** | Meta Training and Inference Accelerator |
| **OS** | Operating System |
| **PE** | Processing Element |
| **PIM** | Processing In Memory |
| **SDK** | Software Development Kit |
| **SRAM** | Static Random Access Memory |
| **UBC** | University of British Columbia |
| **WRAM** | Working Random Access Memory |

# Acknowledgments

Thanks to my supervisor who guided me through the project and thanks to Justin Wong for being an important part of the project. Thanks to my partner for always being there for me and to my parents who have been always supportive.

# Chapter 1

# Introduction

We have witnessed astonishingly rapid progress in machine learning over the past decade. This growth has been driven by advancements in neural networks and our understanding of building specialized neural layers for different applications. While certain applications like *computer vision* and *natural language processing* have received more attention and witnessed frequent changes in their popular algorithms, *recommendation systems* gained attention relatively later in this game.

For a long time, classic recommendation algorithms such as *collaborative filtering* and *knowledge-based* recommendation dominated the field. However, more recently, hyperscalers such as Netflix, Meta (formerly known as Facebook), and Google started exploring the use of neural networks to generate recommendations. Recommendations of content, ads, or products to users directly impact a company's revenue, making it a crucial aspect of their business. Consequently, large companies were hesitant to reveal the cutting-edge methods they were internally employing for recommendation systems. Nevertheless, significant scientific advancements cannot be made without the contribution of a large community of scientists. Thus, less than five years ago, we witnessed the publication of papers on how deep learning and neural networks can be applied in the realm of recommendation systems.

In 2016, Google published a paper on their *wide and deep* learning[6] approach for recommendation, establishing a foundational framework for designing neural networks and employing deep learning in recommender systems. Google's Deep and Wide model remained popular in the field of recommendation systems for

several years until 2019, when Meta (Facebook at the time) published a paper on their Deep Learning Recommender Model (DLRM)[19]. Soon after, they released an implementation of the recommender system in PyTorch and Caffe2 to the public[7]. This enabled researchers to use the available code to conduct experiments on various aspects of the recommender system.

Once an instance of DLRM is trained, it is deployed to handle thousands or even millions of inference queries every day. In 2021, Meta reported that 70% of the AI cycles in their datacenters were dedicated to recommendation inference[10, 14], highlighting the significance of inference performance for these models. Recognizing this problem, we saw an excellent opportunity to leverage the new hardware we recently acquired: the UPMEM Processing-In-Memory (PIM) DRAM modules[24].

Originating in Grenoble, France, UPMEM[23] aimed to bring computation closer to memory by addressing the well-known memory wall problem[26]. Their solution involved adding general-purpose computation units to the working memory of a system. The DIMM modules developed by UPMEM utilize numerous small processing units to perform simple operations in memory, avoiding the data movement overhead between the main processor and memory for each operation. What sets their solution apart is the scalability of computation power with memory size. This allows us to leverage a larger pool of processors for larger memory footprints, ensuring manageable latency.

The remainder of this chapter provides an in-depth exploration of the technologies and concepts employed in this project. We first delve into a comprehensive understanding of the various components of PIM-Rec. Subsequently, we discuss the design decisions we made to construct PIM-Rec and elucidate the challenges we encountered along the way.

## 1.1 Processing-In-Memory

The memory wall problem, also known as the Von Neumann bottleneck, has been a challenge in computing for decades[25]. It arises from the limited bandwidth channel that connects the central processing unit to the main memory. Even with powerful processors and fast memory, achieving high performance for memory-

intensive workloads is hindered by the significant data access latency caused by this bottleneck.

To address this issue, researchers have explored in-memory or near-memory solutions that aim to bring processing closer to the data, thus mitigating the Von Neumann bottleneck. Various in-memory or near-memory architectures have been proposed, some tailored to specific problems such as graph processing[5] or deep learning recommender models[14], while others propose general-purpose solutions that may not be compatible with current hardware[9]. Designing a general-purpose solution that can seamlessly integrate into modern computer systems without compatibility issues is a complex task. In recent years, two general-purpose Processing-in-Memory (PIM) solutions compatible with current systems have emerged: UPMEM DRAM DIMMs and Samsung HBM2 Function-In-Memory (FIM). For the acceleration of PIM-Rec, we employ UPMEM PIM modules, and we will delve into more details about UPMEM PIM in section 3.3.

Samsung has recently published a paper on their HBM2 FIM solution[17]. High-Bandwidth Memory (HBM) refers to memory modules with higher bandwidth compared to conventional memory modules like DRAM DIMMs. HBM modules are optimized for high-performance computing and artificial intelligence workloads and are available in three generations. The second generation, known as Aquabolt, provides up to 1.2 TB/s bandwidth. Unlike UPMEM PIM modules, Samsung HBM2 PIM includes floating-point units. Considering the higher bandwidth of the underlying memory technology in Samsung HBM2, it may offer a more promising PIM solution for this project; however, it is not currently publicly available. While we are aware of recent advancements in the field of PIM, we will utilize the UPMEM PIM solution for this project, as it is accessible to us. Given the architectural similarities between the two solutions, it can be inferred that any performance enhancements achieved using UPMEM are likely to be attainable using Samsung HBM2 as well.

## 1.2 UPMEM PIM Solution

UPMEM [24] provides DRAM DIMMs that are equipped with general-purpose in-order processors, operating at a frequency of 450 MHz. These processors are

referred to as DRAM Processing Units (DPUs). Each 64 MB section of DRAM is accompanied by its own DPU, enabling the computational capacity to scale proportionally with the memory size. Consequently, as the memory size increases, the number of processors also increases. A single DPU consists of 24 hardware threads that can execute in parallel to mitigate the latency caused by memory stalls within the in-order pipeline. The DIMMs are organized in ranks, with two ranks per DIMM, and each rank comprises eight memory chips, housing eight DPUs per chip. For instance, an 8GB memory DIMM would contain 128 DPUs since there is one DPU assigned to each 64MB section. It is important to note that DPUs cannot communicate directly with one another, and any inter-DPU communication must be facilitated through the host processor, which is typically the main CPU of the system. The CPU acts as the orchestrator, responsible for transferring data between the conventional memory and UPMEM DIMMs. Consequently, minimizing the need for inter-DPU communication becomes crucial to avoid imposing additional overhead on the host. It is worth mentioning that the DPU pipeline does not incorporate floating-point units, making DPUs unsuitable for floating-point operations. We have explored this issue in more depth in [3.5]. The architecture of UPMEM DRAM modules is illustrated in Figure 1.1.

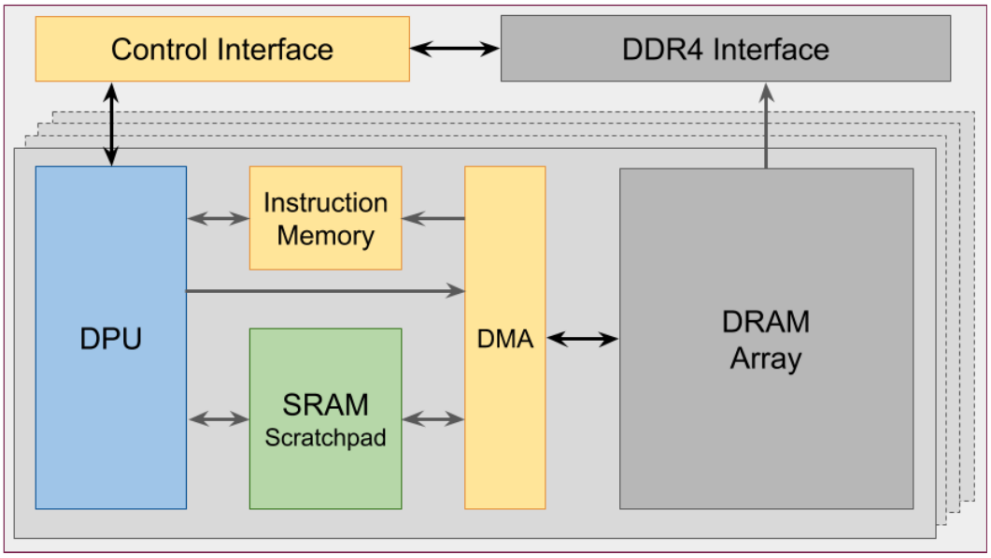



**Figure 1.1:** UPMEM PIM Module Architecture

Each DPU in UPMEM has direct access to a dedicated 64 MB slice of DRAM, referred to as MRAM, as well as a 64 KB SRAM scratch pad. Additionally, each DPU has its own SRAM Instruction Memory. The DMA (Direct Memory Access) module is responsible for handling data transfers between the MRAM and other components. The control interface of UPMEM allows a DPU to operate in either running mode or copying mode. In the copying mode, data is being transferred between the MRAM and the host processor (CPU). It's important to note that these operations cannot occur in parallel with DPU processing. The mode switches are triggered by the DPU program and are not automatically managed by the DMA.

When designing a system using UPMEM PIM modules, it's essential to consider the lack of a standard Memory Management Unit (MMU) in UPMEM modules. In current systems, the MMU in DRAM modules plays a crucial role in memory management, providing functionalities to assist the operating system. One important function of the MMU is to hide data interleaving from the perspective of the operating system, giving the illusion that data is stored sequentially in the DRAM. Data interleaving significantly improves memory bandwidth by enabling parallel reads and writes from different memory banks. In reality, a chunk of data written to a DRAM module is interleaved among different memory banks, with each consecutive byte assigned to a different bank. However, standard DRAM modules can reverse this interleaving through the MMU, allowing the data to be read in its original meaningful form.

In the case of UPMEM DRAM DIMMs, as the data is processed by DPUs, the storage of interleaved meaningless data is not possible. Therefore, using UPMEM DIMMs as the main memory of a system would require significant modifications to the memory management techniques of the hardware. In this project, we utilized UPMEM modules as accelerators rather than as the main memory component.

## 1.3 Embedding Layers

Assuming you want to develop a recommender system for a streaming platform, the effectiveness of your recommendation system heavily relies on how you represent the underlying data. While numerical data such as the number of movies watched or the duration of each movie can be easily represented, there is another

type of data that poses challenges in numerical representation. This type of data includes categorical information like movie names, where the strings of words are not necessarily unique, and movies can share the same name. Additionally, movie names can vary greatly in length. Consequently, categorical data cannot be straightforwardly expressed as numbers. The question then arises: How can we efficiently represent the movies on our platform?

A naive approach to representing movies would be through the use of *one-hot vectors*. In this representation, each vector corresponds to a specific movie. All the vectors have the same length, which equals the total number of movies on the platform. Each element of the vector is set to zero, except for one element, typically assigned a value of 1, indicating which movie the vector represents.

For example, let's consider our platform has "n" movies including "Fight Club" and "Amelie". We can choose any of the elements of the vector to represent "Fight Club" and an arbitrary but different element to represent "Amelie". The one-hot representations of these 2 movies can be seen in figure 1.2. In order to represent features that involve a number of categories, such as the movies a user has watched, we need to use the multi-hot representation. In multi-hot representation, each user will have 1 vector of size "n" in which multiple elements of the vector are non-zero. An example of this multi-hot representation can also be seen in figure 1.2. Is this one-hot or multi-hot vector representation efficient? Can we improve upon it? The flaws of this solution become evident when considering several problems associated with the one-hot system, rendering it impractical for the effective representation of categorical data.

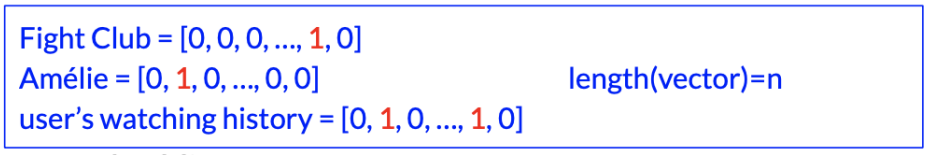



**Figure 1.2:** One-hot and multi-hot representation of categorical features

The first problem arises when dealing with a large number of categories, potentially reaching millions. In the case of a streaming platform, the number of categories, such as movies, can increase rapidly. Consequently, the corresponding

one-hot vectors used for representation become excessively long, demanding significant memory resources and making them challenging to handle. Each time a new movie is added to the platform, all the movie representations must be updated to accommodate the increased length required for representing the newly added movies.

Another crucial problem with the one-hot vector representation is the inability to capture similarities between categories. For instance, movies like ”ET” and ”Star Wars” share much more similarities than ”ET” and ”Pulp Fiction.” However, the one-hot encoding fails to capture such relationships. When applying distance metrics like Euclidean distance, all pairs of vectors are equidistant from each other. This lack of meaningful information about the movies and their properties limits the usefulness of the one-hot representation.

To address the aforementioned challenges, *embeddings* are utilized for representing categorical data. Embeddings are fixed-length vectors, often with dimensions that are powers of 2, such as 16 or 32. The length of the vectors is a decision made by the machine learning engineer based on the requirements of the neural network. Each vector represents a category while capturing the similarities between categories through metrics like Euclidean distance. For a categorical feature with n categories, the *embedding layer* transforms the one-hot system, residing in an n-dimensional space, into a reduced vector space of size D (e.g., D = 32). The elements of this new 32-dimensional vector are not restricted to zeros and ones, but can take any value between -1 and 1. Consequently, an infinite number of categories can be defined within this reduced vector space.

An *embedding table* serves as the data structure that houses all the vectors representing the categories of a categorical feature. An embedding layer may comprise multiple embedding tables, with each table dedicated to a specific categorical feature. The primary role of the embedding table is to map the user-friendly representation of a category, such as a string, to a vector that possesses meaning and utility within the neural network. Typically, embedding layers are placed as the first layer in a neural network to perform this mapping and feed the transformed data into the rest of the network. Without the embedding layer, the neural network would receive incomprehensible strings or vectors as representations of categorical data.

In figure 1.3 we can see examples of embedding tables. We have $C$ embedding tables where $C$ is equal to the number of categorical features our data has. Each table has $D$ columns where $D$ is the size of latent vector space and $N$ columns where $N$ is the number of categories, e.g. movies, of that feature. For each table, the number of categories is going to be different so the $i^th$ table will have $N_i$ rows.

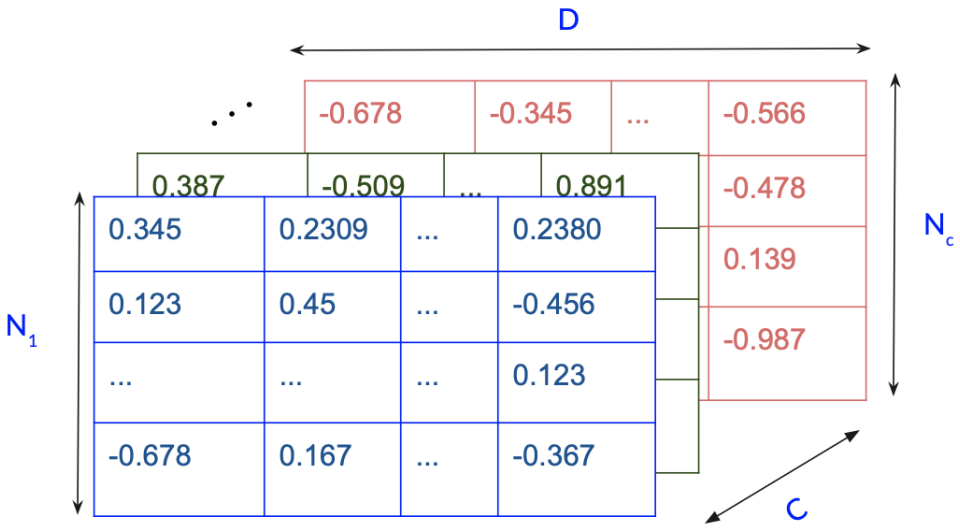



**Figure 1.3:** Example of embedding tables

## 1.4 Embedding Lookups

The operation of retrieving representations of multiple categories from the embedding table and performing a pooling operation on them is commonly referred to as an *embedding lookup*. Typically, a hash function is employed to determine the index of the table row that stores the embedding vector for a specific key. These corresponding rows are then combined through an element-wise operation, such as sum or max pooling. To illustrate this process, please refer to Figure 1.4. For instance, let's consider an embedding table that represents movies. The embedding layer receives a list of movie names, which are passed through a hash function to determine the respective rows in the embedding table. These rows are then retrieved, and their representations are combined, often through element-wise sum.

The resulting output of this embedding lookup is a $D$-dimensional vector that represents the combination of these movies, which can subsequently be fed into the subsequent layers of the neural network.

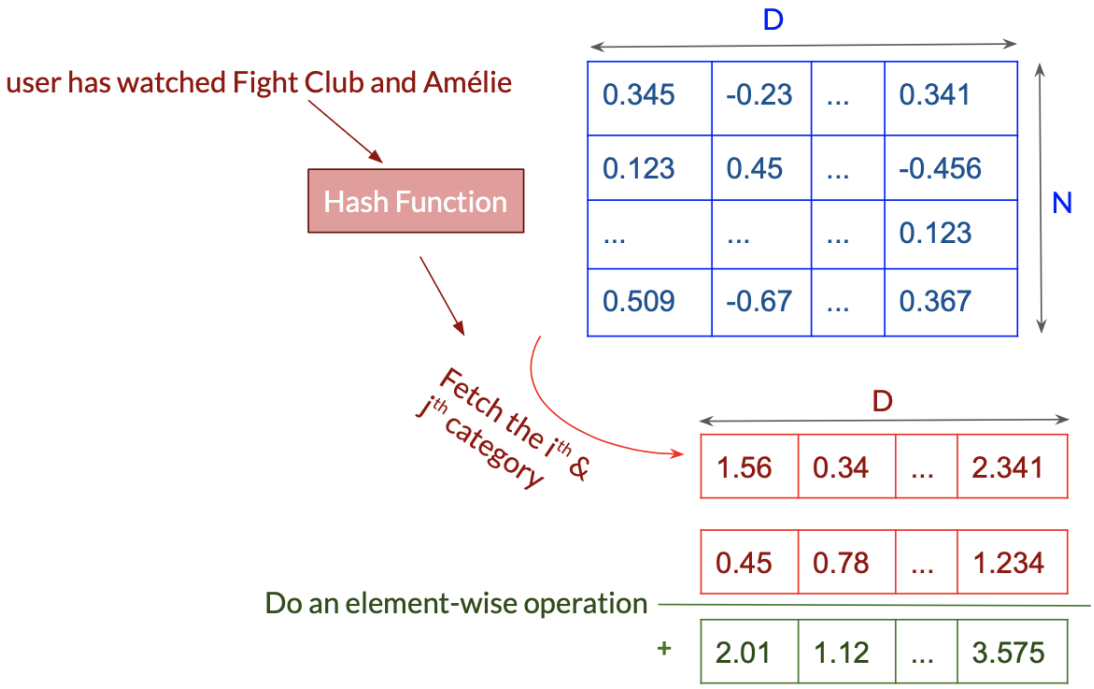



**Figure 1.4:** Illustrain of the embedding lookup operation

It's worth noting that embedding lookups exhibit a non-uniform and unpredictable memory access pattern since the fetched rows can originate from any part of the table. Additionally, it is important to consider that the number of categories can reach millions or even billions, with tens or hundreds of embedding tables potentially present. As a consequence, the embedding layer has a substantial memory footprint, and the lookup operations themselves are memory-intensive. These aspects will be further explored in section 3, where we will analyze their impact on system metrics.

## 1.5 Meta's DLRM

The Deep Learning Recommender Model (DLRM), developed by Meta, has been made publicly accessible to researchers for utilization and study since 2019[19]. DLRM provides a collection of tools that can aid system researchers in examining the performance aspects of the model. Gupta et al. conducted a study on the architectural aspect of DLRM [10] and emphasized the significance of addressing the latency associated with the embedding layer, which represents a memory-intensive component of the inference cycle.

A DLRM instance consists of both dense (numerical) features and sparse (categorical) features. The dense input features are handled by a Multi-Layer Perceptron (MLP), while the sparse features are processed by the embedding layer. The outputs from these components interact at the feature interaction layer. Finally, an MLP is responsible for processing the interacted features and generating a ranked list of recommendations. The overall architecture of DLRM is depicted in Figure 1.5.

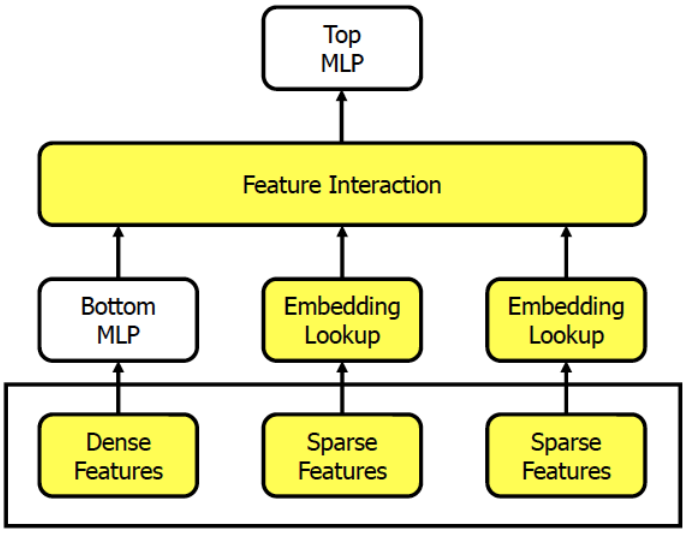



**Figure 1.5:** DLRM general architecture

DLRM can be employed in different input modes. There are three modes avail-

able for the model's input data:

- Dataset: In this mode, the model's input is derived from a real-world dataset. Several publicly available datasets can be utilized with DLRM, including Criteo Kaggle and Terabyte.
- Synthetic: By leveraging knowledge about the characteristics of a real-world dataset and workload, DLRM can be employed in the synthetic mode to generate a workload with identical characteristics. The publicly available DeepRecSys [11], also provided by Meta, can be beneficial in conjunction with DLRM in this mode.
- Random: DLRM has the capability to generate a random workload with uniform distributions, simplifying matters for researchers primarily interested in exploring the architectural and performance aspects of the model.

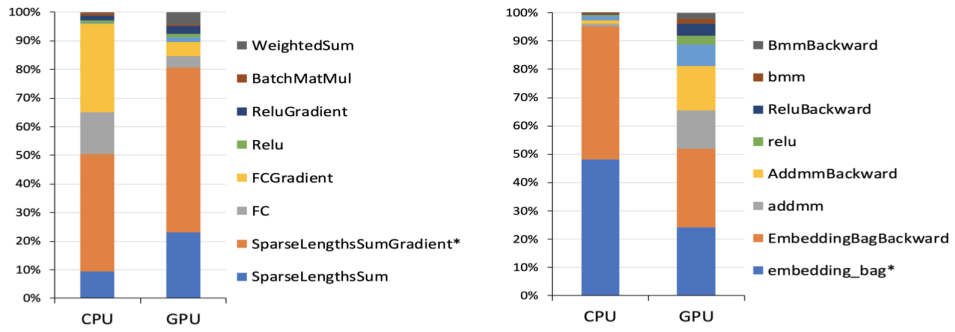
# Chapter 2

# Related Work

## 2.1 DLRM and its architectural Implications

The initial discussion of the algorithmic aspect of DLRM can be traced back to a paper published in 2019 by Naumov et al. [19]. The authors emphasize the inherent difficulties in effective modeling of user behavior and preferences within recommendation systems. To address these challenges and enhance the accuracy and performance of such systems, they propose a deep learning-based approach. The paper provides a description of the architecture and constituents comprising the DLRM model, encompassing key elements such as the embedding layers responsible for user and item representations, as well as the multi-layer perceptron. Despite being an early-stage exploration of DLRM, the authors dedicated a section to the performance challenges presented by the embedding layer. By running performance profilings shown in figure 2.1, they illustrate the importance of embedding layer in inference latency. Figure 2.1 shows performance profilings for both Pytorch and Caffe2 implementations. The blue part at the bottom of the bar charts on the right chart, labeled as "$embedding_bag*$", shows the latency caused by the embedding layer. The same color of blue at the bottom of the chart on the left of figure 2.1 shows the embedding layer latency. The label reads as "*SparseLengthSum*" since Sparse Length Sum (SLS) is another name for the embedding layer and its operations in Caffe2 implementation.

Later on, almost the same group of authors published a paper to provide in-




(a) Caffe2 (b) PyTorch

**Figure 2.1:** Profiling of different implementations of a sample DLRM on a single device[19]

sights into the system performance and architectural considerations behind Meta's DLRM[10]. Their work explores the challenges faced by Meta in building a scalable and efficient recommender system to handle the massive user base and large-scale datasets.

This paper described the challenges in designing and optimizing each component of DLRM, including issues related to scalability, efficiency, and model accuracy. it discusses techniques such as data parallelism, model parallelism, and model compression to address these challenges and improve the performance of the recommendation system. Yet, These techniques are mostly useful for fully-connected layers within MLPs. The issue of embedding layer latency and memory footprint is left unsolved. In figure 2.2 we can see the inference latency for 3 different production-scale DLRM models. All the models are using the same batch size for better comparsion. The latency breakdown on the left side of figure 2.2 shows up to almost 80% of inference latency can be caused by embeddings.

By reading this paper, we realize how important meeting throughput System Level Agreements (SLAs) are since DLRM is a user-facing application. We earn insights into the deployment and serving infrastructure employed by Meta to handle the high-throughput demands of personalized recommendations. We also see the use of distributed systems, caching, and load balancing techniques to achieve low latency and high throughput for recommendation requests can offer consid-

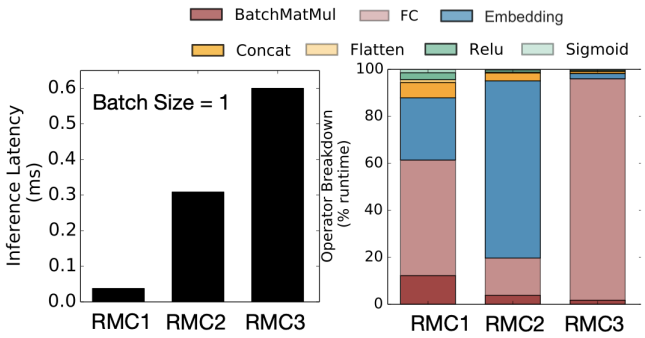



**Figure 2.2:** Inference cycle latency for 3 production-scale instances of DLRM (left) Breakdown of the inference cycle latency for each instance[10]. [10]

erable performance improvements. Yet, they do not fully address the embedding latency problem for embedding intensive models.

## 2.2 Recommender Models and Processing In or Near Memory

Considering what we have learned so far about DLRM and the optimization techniques employed by Meta researchers, there is limited scope for further improving the performance of dense layers. However, sparse embedding layers present a significant opportunity for performance optimization [10]. The workload exhibits low locality, with only temporal locality observed due to the reuse of popular items or categories. Therefore, caching can provide moderate enhancements [4, 10, 14].

The distinctive memory access patterns and the high latency incurred within each inference cycle have motivated researchers to develop a specialized Near-Memory-Processing solution known as RecNMP [15]. RecNMP proposes a commodity DRAM module utilizing DDR4 DIMMs to enable the processing of embedding pooling operations near memory. Their approach assumes a CPU-centric system for inference, and several hardware-software co-optimizations have been implemented. Simulations demonstrate a 9.8X reduction in memory latency and a 4.2X improvement in throughput. These optimizations include table-aware packet scheduling, memory-side caching, and hot entry profiling. RecNMP primarily

targets Meta datacenters, and considerable effort has been devoted to addressing co-location issues. The researchers have explored strategies for the optimal co-location of models on servers and the distribution of queries among replicas of the same model. However, it is important to note that RecNMP has only been evaluated through simulations, and To the best of our knowledge, the actual hardware has never been built.

In contrast, our solution offers the unique advantage of processing data directly in memory without requiring data movement between the DRAM module and a specialized processor like an FPGA. PIM-Rec leverages a real system built on general-purpose hardware, making it more versatile compared to solutions designed specifically for this class of models. Its adaptability to various memory-intensive tasks and workloads makes it easily deployable in datacenters, harnessing the benefits of the proposed general-purpose hardware.

Other researchers have explored the use of near-memory processing for embedding lookups. TensorDIMM proposes a vertically integrated hardware/software co-design for DIMM-based DRAM modules explicitly tailored for processing embedding lookups [16]. Their modules function as a remote memory pool, working in conjunction with a GPU-centric system that utilizes GPUs for inference. Their simulated hardware design demonstrates a performance improvement ranging from 6.2X to 17X. However, as we will discuss in section 3.5, a CPU-centric approach is preferred for recommender model inference, as a GPU-centric system may struggle to meet SLA latency standards while consuming more energy. TensorDIMM's solution is based on the assumption that embedding tables' maximum width is 64, limiting its generality. In contrast, PIM-Rec does not make such assumptions and offers a generalized solution validated by real experiments.

The same research team from KAIST has proposed *Centaur* as a hardware-software co-designed solution to enhance the performance of Deep Learning Recommender models, focusing on both dense and sparse features [13]. Centaur proposes a chiplet-based architecture that colocates the CPU and FPGA on the same chip, enabling higher bandwidth between them. This design allows direct memory access for both the CPU and FPGA, resulting in reduced latency when fetching embedding vectors from memory. By addressing the low compute bandwidth of CPUs in CPU-centric inference for recommender models, Centaur tackles the chal-

lenge faced by DLRM instances with smaller embedding layers but high compute intensity in the fully-connected layers of the MLP. The FPGA component in this hybrid design complements the limited computing power of the CPU.

However, deploying specialized chiplets dedicated solely to a specific class of applications, such as deep learning recommender models, in datacenters is not practical. Additionally, programming and maintaining FPGAs at scale in datacenters pose their own challenges, which hyperscalars are often reluctant to undertake.

In summary, Centaur proposes a hardware-software co-designed solution to enhance Deep Learning Recommender models by considering both dense and sparse features. Its chiplet-based architecture colocates the CPU and FPGA, facilitating improved bandwidth and lower latency for memory access. While Centaur addresses the low compute bandwidth issue of CPUs in recommender models, deploying specialized chiplets in datacenters and managing FPGAs at scale pose practical challenges for hyperscalars.

## 2.3 Meta Training and Inference Accelerator

In 2023, Meta introduced their own specialized DLRM (Deep Learning Recommendation Model) inference accelerator, known as the Meta Training and Inference Accelerator (MTIA)[1]. This accelerator was purposefully designed and fabricated as an ASIC (Application-Specific Integrated Circuit) to cater to the specific requirements of recommender inference and training within Meta datacenters. MTIA was seamlessly integrated into PyTorch, resulting in an optimized full-stack solution.[1, 8]

MTIA is equipped with a grid of processing elements (PE) capable of accessing both on-chip and off-chip memory. To efficiently manage compute and memory resources, a dedicated control subsystem operates the accelerator's firmware, maintaining a dedicated interface with the system's main processor and orchestrating tasks executed on the accelerator.

The on-chip memory of MTIA comprises 128-MB of SRAM, while the off-chip memory consists of 128-GB of DRAM. The use of on-chip memory primarily focuses on frequently accessed data to ensure higher bandwidth and lower latency. Each chip contains 64 PEs, which can communicate with each other and effectively

process floating-point operations.[8]

MTIA shares many similarities with PIM-Rec in terms of addressing similar problems by bringing compute closer to memory, which aids in resolving bandwidth-related issues. Notably, MTIA exhibits remarkable performance with low and medium complexity DLRM instances, offering up to 2.84X speedup compared to GPUs for medium complexity instances of DLRM.

However, it is essential to acknowledge that PIM-Rec offers a more generic solution that can be extended to address embedding lookups not only for recommender models but also for other types of deep learning models utilizing embeddings. Additionally, the estimated average cost per unit for UPMEM DIMMs is considered lower compared to MTIA.

Overall, MTIA represents a significant advancement in addressing recommender inference and training needs within Meta datacenters, showing substantial speedup for specific DLRM instances. Nonetheless, the broader versatility and cost-effectiveness of PIM-Rec's approach make it an attractive option for various deep learning models using embeddings.

# Chapter 3

# Workload Characteristics

For our initial experiments, we utilized three instances of Meta's Deep Learning Recommender Model (DLRM)[19], which were pre-trained with the Criteo Kaggle[3] and Terabyte[2] datasets included in the MLPerf inference benchmark suite[22]. MLPerf provides standardized benchmarks covering various tasks, including computer vision and recommendation, for evaluating underlying systems. It is widely accepted and used by the community and offers both training and inference workloads.

In this phase of the study, our primary focus was to understand the DLRM workload and the characteristics of the systems it runs on (see Section 4.1 for details). To evaluate the performance of PIM-Rec, we employed Meta's DLRM with randomly generated input data, which allowed us to have flexibility in workload generation and model architecture (refer to Chapter 5 for more details).

PIM-Rec leverages Meta's DLRM as the model to be served, benefiting from its ability to generate workloads with various architectural and runtime parameters. To gain insights into the nature of the workload handled by PIM-Rec, we conducted experiments performing inference on the pre-trained DLRM instances provided by the MLPerf inference benchmark. The model we utilized[18] was trained using the Criteo Kaggle dataset[3]. Our main interest was to observe the memory, cache, and CPU characteristics of this workload.

By analyzing the cache usage of the workload, we observed high miss rates, particularly when the inference batch size was large. Larger batch sizes are gen-

erally preferred for inference tasks as they can offer higher throughput, assuming the SLA latency criteria are met. However, due to the high cache miss rates in this workload, we were unable to fully leverage the potential performance improvements that larger batches could provide by utilizing the system's compute resources more efficiently. Figure 3.1 illustrates the LLC cache miss rates, reaching as high as 84.26% for larger batch sizes, such as 2000. A similar pattern of higher miss rates was observed for the L1D cache, with even small batch sizes experiencing high miss rates (see Figure 3.2).

Deep Learning Recommender Models are generally not compute-intensive. To monitor and profile how the CPU is being utilized by these models, we used the same DLRM model to observe the Instructions Per Cycle (IPC). The experiments were conducted on a machine with 4 CPU cores, and the observed IPC was as low as 0.44, which is only 10% of the system's highest potential IPC (see Figure 3.3). These experiments also utilized the DLRM model trained with the Criteo Kaggle dataset[2, 18]. By juxtaposing the cache performance of the model with its CPU usage, it becomes evident how memory intensity of the workload impacts performance and highlights the significance of addressing it.

We were also curious to observe how adding parallelism would affect DLRM's Inference Latency. Our primary interest was to determine if highly parallel computing would help decrease latency or if the memory bottleneck would dominate the effect of parallelism. To investigate this, we employed various numbers of CPU cores to handle the same inference job and measured the latency. As depicted in Figure 3.4, increasing the number of CPU cores did lead to a reduction in latency, but it was not a linear relationship. In fact, the latency decrease was far from linear, indicating that adding more CPU cores to this workload does not provide significant benefits.

Upon analyzing the results from Figure 3.4, we became eager to understand the reasons behind this phenomenon and the minimal latency change between different numbers of cores. We delved into the stack trace while processing the Embedding Layer within an inference cycle. Upon further examination of the operations performed by each core during embedding lookups, we observed that the work was distributed across multiple CPU cores. However, some cores encountered page faults while others did not. Cores experiencing page faults exhibited variances and

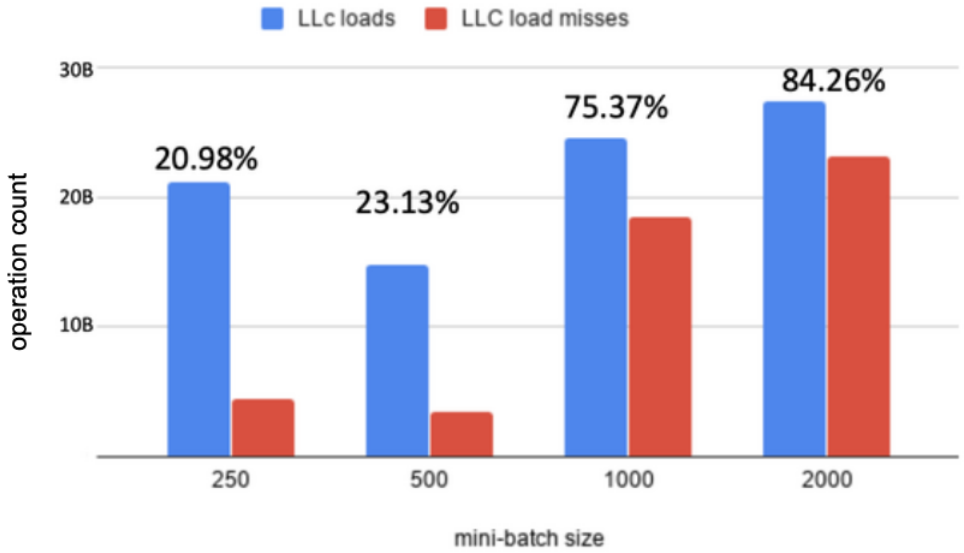



**Figure 3.1:** LLC miss ratio for different batch sizes for a DLRM instance trained on Criteo Kaggle dataset[3] and included in MLPerf benchmark suite[18]- The percentages on top of bars indicate cache miss rate

spent the majority of cycles handling them, as illustrated in Figure 3.5, Figure 3.6, and Figure 3.7 for cores without page faults and with page faults.

These findings indicated that the CPU cores were indeed contending for memory and spent a significant amount of time stalling for memory access. As a result, neither increasing the CPU core performance nor adding more CPU cores proved to be effective in scaling up the performance of embedding lookups. The minor performance gain shown in Figure 3.4 merely allowed other abstracted operations before and after the actual embedding lookup operation in DLRM to run faster. To achieve effective and scalable acceleration of embedding lookups, it is imperative to address the memory-wall problem.

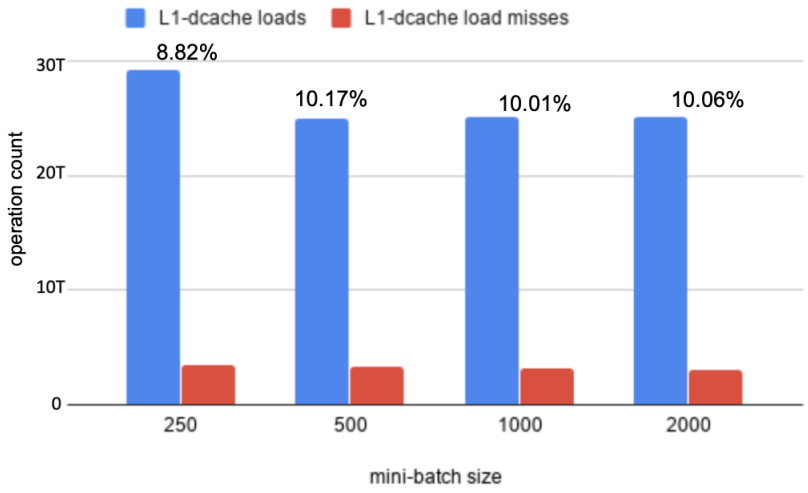



**Figure 3.2:** L1D miss ratio for different batch sizes for a DLRM instance trained on Criteo Kaggle dataset[3] and included in MLPerf benchmark suite[18]- The percentages on top of bars indicate cache miss rate

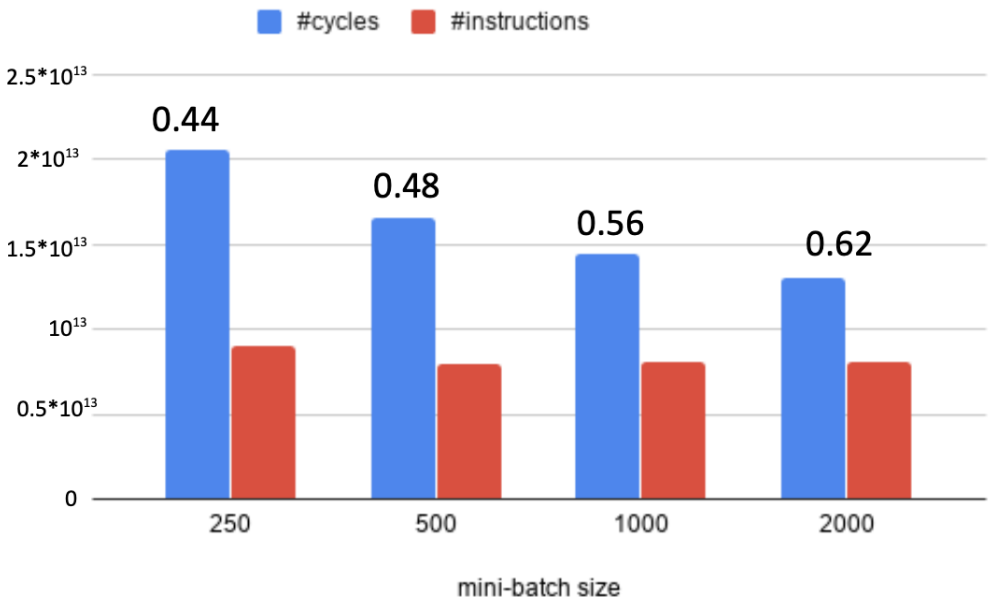



**Figure 3.3:** IPC of a 4 core machine for different batch sizes for a DLRM instance trained on Criteo Kaggle dataset[3] and included in MLPerf benchmark suite[18]

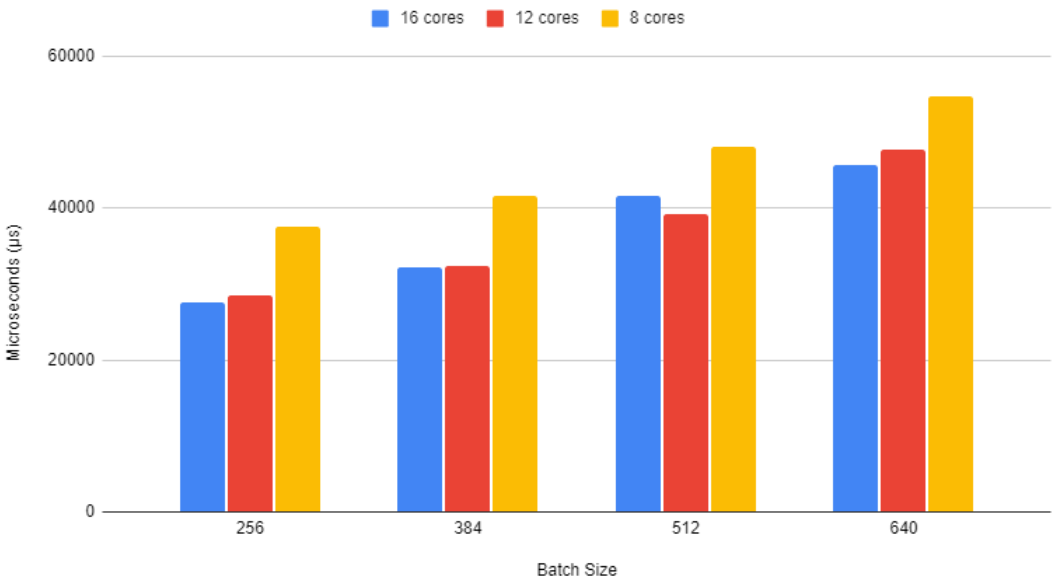



**Figure 3.4:** Latency change with different number of cores for a DLRM instance trained on Criteo Kaggle dataset[3] and included in MLPerf benchmark suite[18]

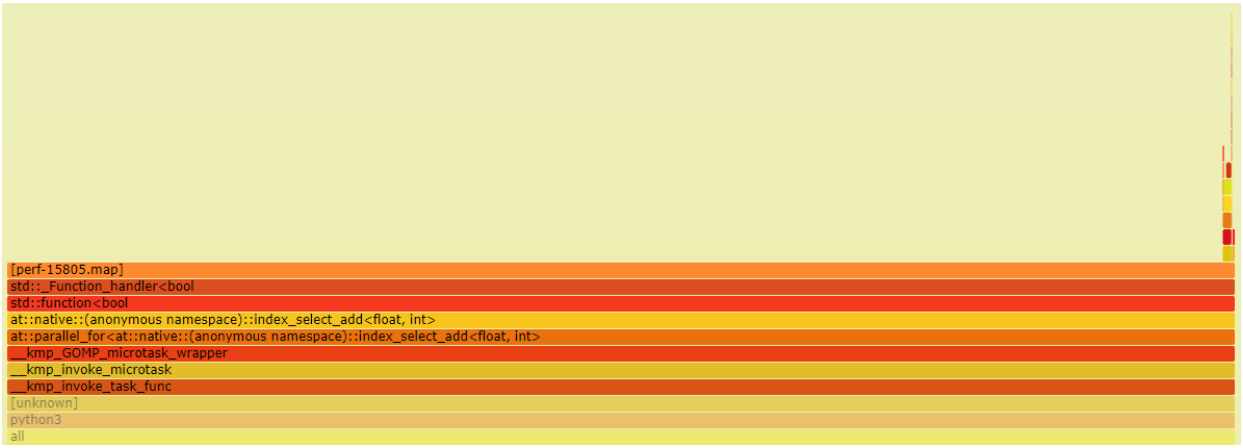



**Figure 3.5:** Stack trace graph for a process without page faults during embedding lookup for a DLRM instance trained on Criteo Kaggle dataset[3] and included in MLPerf benchmark suite[18]

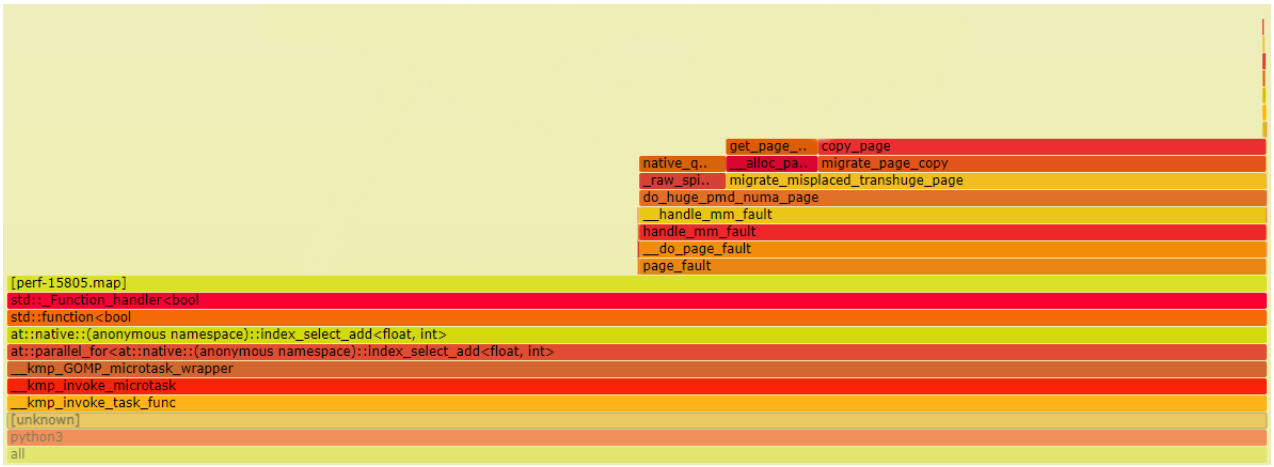



**Figure 3.6:** Stack trace graph for a process with page faults during embedding lookup for a DLRM instance trained on Criteo Kaggle dataset[3] and included in MLPerf benchmark suite[18]

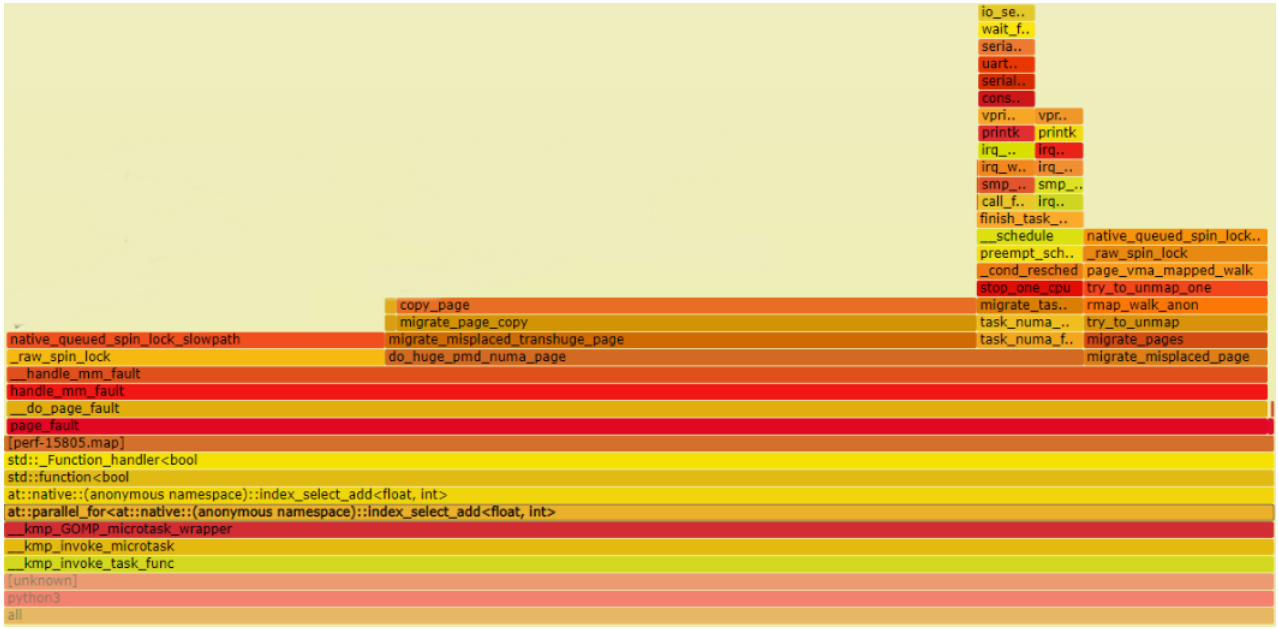



**Figure 3.7:** Another Stack trace graph for a process with page faults during embedding lookup for a DLRM instance trained on Criteo Kaggle dataset[3] and included in MLPerf benchmark suite[18]

# Chapter 4

# PIM-Rec Design

PIM-Rec is specifically designed for a CPU system equipped with PIM-enabled DDR4 DIMM DRAM modules, which act as accelerators for the embedding layer. The workload is generated using Meta's DLRM and is then sent to the PIM module by the host CPU. Once prepared, the lookup results are subsequently transferred from the accelerator back to the system's main processor. PIM-Rec encompasses two primary phases within its design:

- **Pre-processing phase:** In this phase, the embedding tables are copied to the PIM modules and remain there for the duration of the system's operation. Since this operation needs to be performed only once per model instance, we neither optimize nor evaluate the overhead of this operation.
- **Lookup phase:** During each inference cycle, a query comprising a batch of multiple lookups is directed to the PIM module. The PIM module performs the necessary computations, and when the results are ready, they are copied back to the host CPU to be fed into the subsequent layers of the neural network.

In the following chapter, we will provide a detailed explanation of each phase of the PIM-Rec system's design.

## 4.1 CPU-Centric Design

GPUs have revolutionized the field of deep learning, playing a pivotal role in driving Machine Learning to its current state of excellence. Their high-throughput and remarkable parallel computing capabilities have been instrumental in advancing the field, making GPUs the go-to choice for training deep learning recommendation models. However, when it comes to inference, GPUs may not always be the most optimal solution.

In customer-facing applications heavily reliant on inference, minimizing latency is of utmost importance [12]. The objective here is not necessarily to optimize the experience for each individual user to its maximum potential, but rather to provide an acceptable experience for the majority of users. Thus, in such applications, the goal is to optimize for latency-bound throughput. GPUs excel in scenarios where highly parallel workloads need to be optimized for throughput. However, they may not be the best choice for minimizing latency or achieving latency-bound throughput [12]. Moreover, the cost factor associated with GPUs should also be taken into consideration, particularly when evaluating their value at scale. In hyperscale environments, the use of GPUs for handling user requests can significantly increase the Total Cost of Ownership (TCO). Consequently, CPU-centric systems are often preferred for performing inference on Recommender Models, even though these models heavily rely on GPUs for training[8].

## 4.2 Minimizing Implementation Overhead

During the initial phases of developing PIM-Rec, we quickly realized the criticality of the implementation specifics in our solution. To comprehend the significance of this aspect, it is essential to examine the software stack that our lookup queries traverse. Figure 4.1 illustrates the software stack that each lookup query undergoes upon its arrival. Similarly, the response to the query traverses the same stack in reverse order. The programming language utilized at each stage is indicated in blue at the bottom of each corresponding box in Figure 4.1.

Python is renowned for its ease of use and capability to implement intricate logic with just a few lines of code, liberating developers from delving deeply into the underlying mechanisms. It has gained immense popularity within the Machine

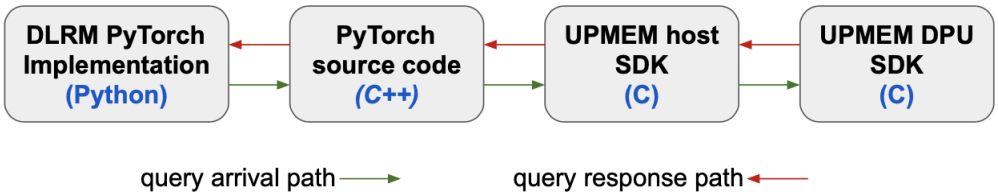



**Figure 4.1:** The software stack each lookup query goes through

Learning (ML) community, and major ML frameworks like PyTorch and TensorFlow are predominantly built on Python. However, Python's memory management does not grant users fine-grained control over memory, unlike languages such as C or C++. Consequently, the memory allocated in Python cannot be seamlessly reused in C or C++ without incurring problems and overhead.

DLRM's lookup queries in PyTorch are essentially memory segments allocated by Python. In the context of PIM-Rec, the UPMEM SDK acts as the consumer of this memory, enabling control of DPUs and data transfer to or from them. Several methods and tools exist to aid in allocating memory that can be passed from Python to C. One such tool is the Ctypes library, which allows users to allocate C-compatible memory within Python. It also facilitates the importation of C functions into Python code and their invocation by passing C-compatible arguments. However, a crucial point to consider is that data stored in Python structures, such as Python lists or Tensor instances, must be copied to the C-compatible variable provided by Ctypes. Memory copies can be expensive, particularly in our memory-intensive workload.

The initial implementations of PIM-Rec, which utilized the Ctypes library, exhibited an average overhead of 3X when transferring queries from PyTorch to the UPMEM SDK compared to performing lookups on the CPU as DLRM does by default. This necessitated finding an alternative approach to query handling that eliminates the need for memory copies. Thus, we delved deeper into PyTorch's implementation to uncover its inner workings.

PyTorch offers a Python interface, but many of its core functionalities are implemented in C++. The Python bindings provided within the code generate the Python interface when the C++ code is compiled. This duality of Python and

C++ enables PyTorch to be user-friendly while maintaining high-performance efficiency. Upon inspecting the implementation of embedding layers in PyTorch, we discovered that the data structures and functionalities are all implemented in C++ and wrapped in the Python API. This presented an opportunity for us to leverage the available queries in C++ and pass them to the UPMEM SDK since C++ pointers can easily be reused in C.

Consequently, rather than integrating calls to the UPMEM SDK in Python code, we incorporated them directly into the C++ source code of PyTorch. This decision allowed us to circumvent memory copies and significantly reduce the overhead introduced by PIM-Rec during lookup query processing. However, this approach came with its own trade-offs, namely the necessity to recompile PyTorch and build our custom version specifically tailored for PIM-Rec, rather than utilizing the pre-built version commonly installed through package managers like pip. Nonetheless, this trade-off proved worthwhile, as it resulted in substantial performance benefits for the end-users. By adopting this strategy, we successfully minimized query transfer overhead and harnessed the true performance advantages of offloading lookups to PIM modules.

## 4.3 Pre-processing Embedding Tables

As mentioned in the background section about UPMEM PIM modules, they cannot be used as the system's main memory. Therefore, in PIM-Rec, the Embedding Data needs to be copied to the PIM module from the main memory. Depending on the scenario, we use either a pre-trained model or generate a random one on the fly. After loading the entire model to the main memory, PIM-Rec moves the embedding tables to the MRAM memory. The embedding table data is copied only once and remains in MRAM as long as the model is up and running. The number of DPUs (Data Processing Units) to which the embedding tables are copied depends on the model architecture, i.e., the number of embedding tables and the width of each table. In real-world recommender models, re-training typically occurs at most once or twice a day. Each time the model is re-trained and the embedding data is modified, the pre-processing phase can be executed again to load the new data to PIM memory.

The way embedding data is distributed and placed on PIM memory directly affects the performance we can achieve from DPUs. To optimize performance, we need to place data on PIM memory in a manner that maximizes the utilization of available parallelism. Therefore, during the pre-processing phase, the embedding tables are broken down into columns, and each DPU will receive one or multiple columns of the Embedding Table until its capacity is full. Further details on this process will be provided in the next section. At this phase, a mapping of Embedding data to DPUs will be generated and maintained by the host processor, allowing it to direct queries to the appropriate DPUs later on.

## 4.4 Embedding Data Layout


**Figure 4.2:** The UPMEM DIMM which includes 64 DPUs on each side of the DIMM (rank) and 128 DPUs in total on each DIMM that includes 2 ranks[24]

Each UPMEM DIMM consists of 2 ranks. As depicted in Figure 4.2, each side of a DIMM is called a rank and contains 64 DPUs spread over 8 chips. Each chip is equipped with 512 MB of DRAM memory. Consequently, each rank has 4GB of memory, organized into 64 small sections, with 64 MB per section. Each 64 MB section has its own processor, known as a DPU.

Within each DPU, there is a 64 MB share of DRAM called MRAM. However, for storing Embedding Tables, only 56 MB of MRAM is utilized, leaving some buffer space for storing the embedding lookup results. This ensures that the CPU can read and access the results easily. Additionally, some MRAM memory is required for copying the lookup queries from the host. Considering that tables are

broken down column-wise, each DPU receives the maximum number of columns that require less than 56 MB of storage.

The element-wise operation employed in DLRM is summation. The key characteristic of element-wise summation is that it can be performed independently on the elements of a vector. For instance, when summing two vectors, such as $A = \{0.89, -0.67, 0.54\}$ and $B = \{-0.98, 0.75, 0.45\}$, the outcome of $a_0 + b_0$ is completely independent of $a_1 + b_1$. Exploiting this independence, we can assign each column of the embedding table to a different DPU. This approach enables the accommodation of large embedding tables with minimal inter-DPU communication. Otherwise, the 64 MB of MRAM memory allocated to each DPU would be insufficient for conventional table sizes. By breaking down the tables column-wise, we ensure that no additional post-processing is required on the host processor side to calculate the final results from the partial results generated by the DPUs.

To provide a clearer illustration, consider an embedding table of size 20. If we break it down row-wise, the first 10 rows will be assigned to $DPU_0$, while the last 10 rows will be assigned to $DPU_1$. Now, a lookup query arrives that requires element-wise summation of rows $2, 7, 12$. We can easily sum up rows 2 and 7 in $DPU_0$, but row 12 is stored in $DPU_1$, and DPUs cannot communicate with each other. Consequently, we need to sum up rows 2 and 7, copy the result from $DPU_0$ to the host CPU, copy row 12 from $DPU_1$, and utilize the main CPU to perform the summation of $T_2 + T_7$ with $T_12$. However, if we break down the tables column-wise, in most cases, all the rows of $col_i$ will be stored in $DPU_i$. Therefore, when the element-wise summation for $col_i$ of a query is computed in the DPU, it represents the final result for that element of the vector. On the host side, we simply need to ensure the elements are arranged in the correct order when copying them from DPUs to the host.

The distribution of tables among DPUs and ranks varies based on the number of columns in the embedding tables. It is known that the number of columns will be a power of 2, ranging from 8 to 128. The most common table lengths are 32 and 64. Figure 4.3 illustrates how the tables are distributed among the DPUs according to the table width.

An idea we explored to maximize throughput when the Embedding Layer is not extensive and occupies less than 50% of the PIM memory was to replicate the

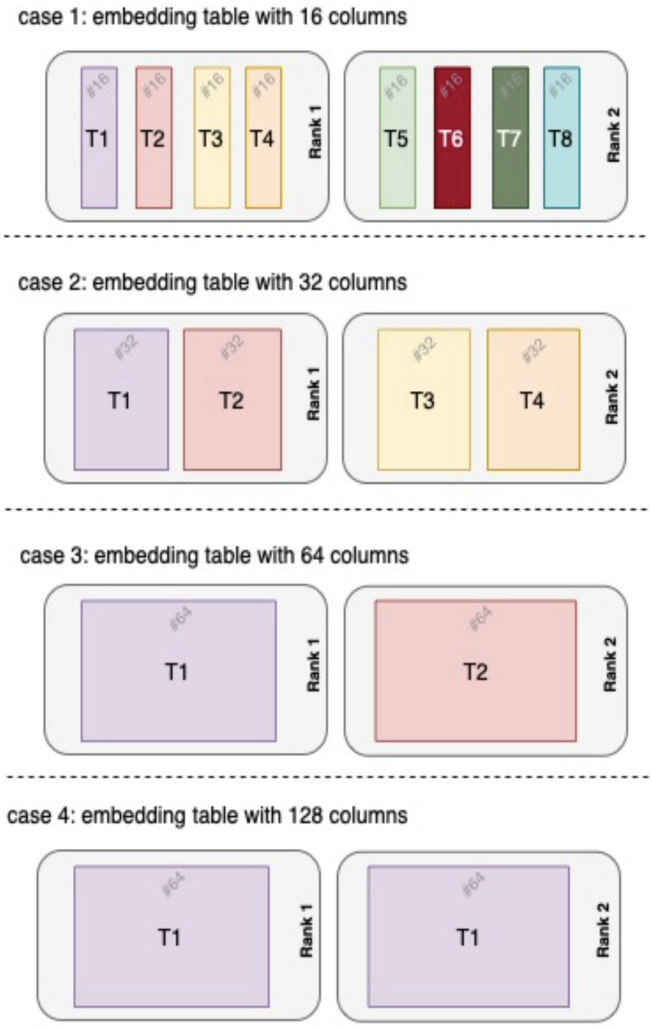



**Figure 4.3:** Data Distribution Layout for Different Embedding Table Column Size

embedding tables. In this scenario, instead of having one DPU responsible for a portion of the Embedding Data, two or more DPUs would share the same portion. When a query arrives at the PIM memory, the replicated DPUs would receive only a subset of the batch. Consequently, a batch of lookups would be divided among the replicas, leveraging a larger number of DPUs and achieving higher throughput

through increased parallelism. However, this approach introduces overhead to the host processor. The host processor in this CPU would be responsible for sorting the results copied from the replica DPUs. The associated overhead cost could potentially outweigh the speedup gained from offloading the lookups to DPUs. Therefore, we did not pursue this idea.

## 4.5 PIM Embedding Lookups

On each inference cycle, a query comprising a batch of queries is copied to the PIM memory. Utilizing the mapping available to the host processor, each part of the query is copied to the corresponding DPU or DPUs responsible for storing that portion of the embedding data. Once the entire query is correctly placed in the PIM memory, the DPUs are launched, and they initiate the processing of the query, computing the lookup results.

The queries received by the DPUs consist of two 1D arrays: the *indices* array, which stores all the indices needed for the lookup batches, and the *offsets* array, which indicates the starting points of each lookup batch. Initially, the query was in a 2D array format, with each individual lookup operation represented as a 1D array of indices to fetch for that operation. The query itself contained a batch of these 1D arrays, as each lookup query combined several independent lookup operations. For instance, a single query might consist of 64 independent lookup operations, with each operation encompassing 32 indices, resulting in the lookup of 32 rows from the table. The indices and offsets arrays observed in the DPUs represent the flattened representation of the original 2D query array. Figure 4.4 provides an illustration of how the query arrives in the DPUs.

To mitigate the latency caused by memory reads, each DPU employs 12 tasklets to fill its pipeline efficiently. Each tasklet is responsible for handling one batch of lookups at a time and generating one final result for that batch across the subset of columns of a table assigned to its DPU. Once a batch is completed, the tasklet proceeds to the next one in a round-robin fashion. Figure 4.5 illustrates how the lookup operations are divided among the tasklets within a DPU, effectively harnessing the available parallelism.

The computed results from the tasklets are stored in a 2048-byte buffer in

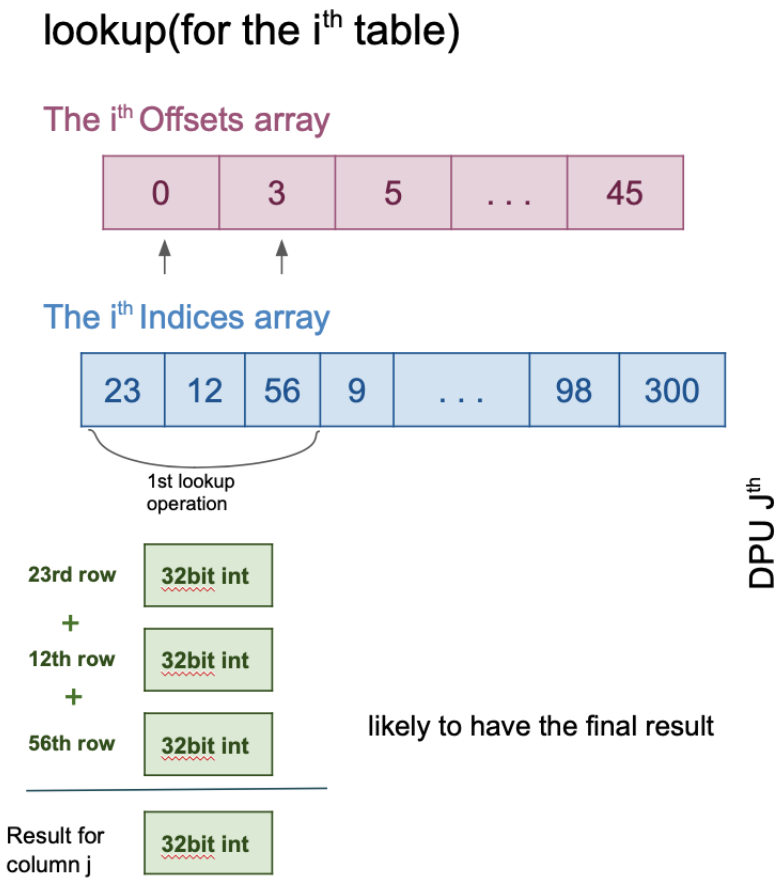



**Figure 4.4:** Illustration of a lookup query on the $i^{th}$ table from the perspective of the $j^{th}$ DPU, which stores the $j^{th}$ column of the $i^{th}$ table.

WRAM. The size of each DMA (Direct Memory Access) transfer between WRAM and MRAM can range from 8 to 2048 bytes, with the highest memory transfer throughput achieved when the largest size is used. Once the results accumulated by the tasklets reach 2048 bytes or the entire query has been processed, the computed results are copied from WRAM to MRAM, where they can be accessed by the host.

Each rank of DPUs corresponds to a CPU thread responsible for handling the copying to or from that rank. These threads are initiated during the pre-processing

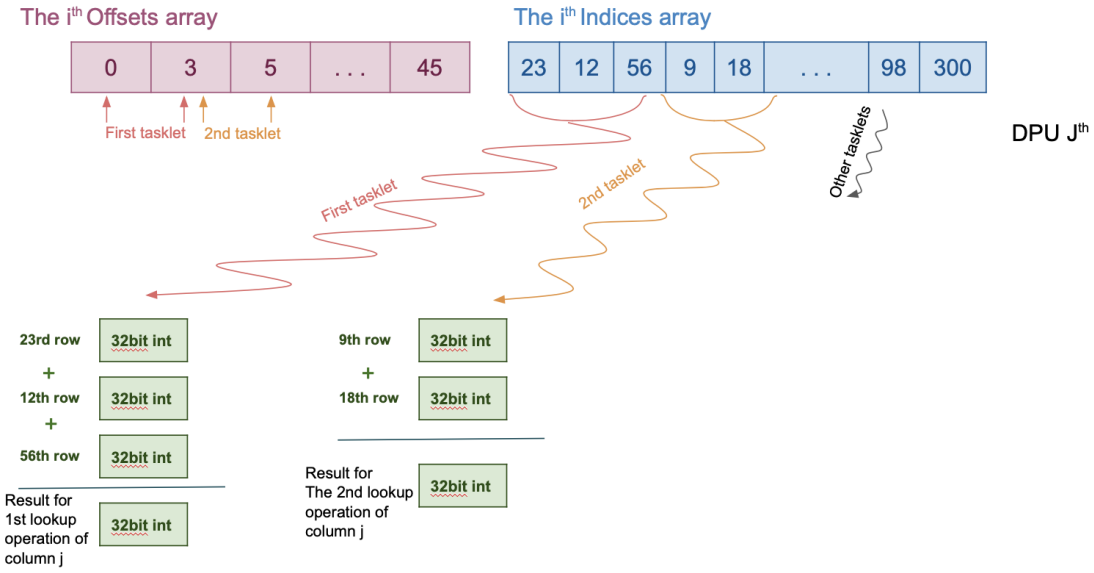



**Figure 4.5:** Illustration of how a lookup query on the $i^{th}$ table is distributed among the tasklets of the $j^{th}$ DPU to utilize the parallelism offered by the DPUs.

phase when the DPUs are allocated. Additionally, a rank-specific host thread can handle the post-processing of the results generated by that particular rank. This means that the ranks can operate asynchronously, leveraging both the parallelism available at the host side and the parallelism offered by the PIM module. The post-processing performed by the rank-specific host threads involves reformatting and arranging the results to prepare them for input to the remaining layers of the neural network. Once all processing is completed, PIM-Rec sends a signal, marking the end of the forward pass through the Embedding Layer for that inference cycle. The signaling and orchestration of host threads are managed by an additional thread on the host, whose role is to continuously poll the rank-specific threads.

# Chapter 5

# Experimental Results

After presenting the information about PIM-Rec, we will now examine its performance in action. The implementation of PIM-Rec can be accessed in a GitHub repository [21]. The main repository contains the host code and DPU code, both of which are written using UPMEM SDK. Additionally, this repository contains several submodules. Among these submodules, one is a fork of Meta's DLRM implementation repository [20], and the other is a fork of Pytorch, which includes our custom Pytorch version. Users have the option to recompile and install this custom Pytorch before running PIM-Rec.

Once we completed the end-to-end implementation of PIM-Rec, we subjected it to various workloads with diverse characteristics to evaluate its performance. Our primary focus was on systemic aspects, such as cache and processor usage.

To assess the impact of offloading embeddings and embedding lookups to DPUs, we compared the system performance using the default Pytorch implementation (which utilizes the CPU) with PIM-Rec. During the experiments, we examined the following metrics:

- **Speedup:** We aimed to determine the speed improvement achieved by offloading embedding lookups to DPUs. To do this, we measured the inference cycle latency and calculated the ratio of defualt Pytorch latency to PIM-Rec latency for the same workload.
- **IPC:** In order to assess processor efficiency, we monitored both CPU and

DPU IPC for both baseline Pytorch and PIM-Rec implementations.

- **Cache metrics:** We analyzed the difference in cache usage between the two implementations by studying the cache hit rate of CPU LLC and L1D.
- **Latency breakdown:** Understanding the distribution of latency across each stage of the lookup pipeline in PIM-Rec provided insight into scalability and potential bottlenecks.

In the subsequent sections of this chapter, we will delve deeper into each of these metrics and explore the performance of PIM-Rec under various circumstances.

## 5.1 Speedup

The primary concern about PIM-Rec is its speed compared to the baseline Pytorch implementation for DLRM inference. To address this, we measured the latency of an inference cycle for the same workload using both lookup methods. We measured end-to-end latency which encompasses the entire process of handling one inference query through all the layers of the DLRM instance, including both the latency of the embedding layer and the MLP layers. We chose to measure end-to-end latency rather than solely focusing on the embedding layer latency for two main reasons:

1. End-to-end latency provides a more comprehensive understanding of overall performance, which is our main concern.
2. The latency of different layers is not entirely independent. How one layer utilizes system resources can impact the completion time of other parts of the neural network.

To ensure high accuracy in our measurements, we conducted inference on the same DLRM instance 100 times for each lookup implementation and then calculated the average of those 100 inference cycles as the latency for that specific lookup implementation and workload.

We initially aimed to determine the extent of speedup offered by PIM-Rec and accordingly selected a favourable workload for PIM-Rec based on the system

architecture. Figure 5.1 illustrates the observed speedup of up to 9.61X compared to baseline CPU implementation for this workload, the characteristics of which are detailed in Table 5.1.

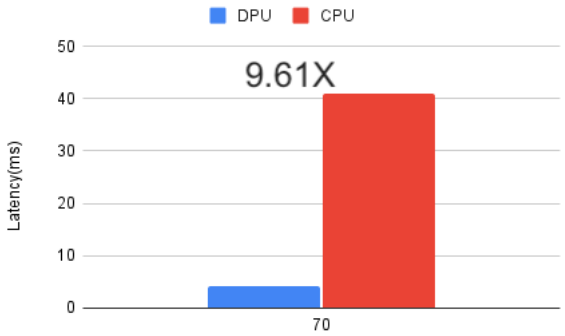



**Figure 5.1:** Inference cycle speedup of PIM-Rec for an favourable workload shown in table 5.1

| # DPUs | emb data per DPU | query size | # emb tables | emb table size |
|---|---|---|---|---|
| 4480 | 400 KB | 4 KB | 70 | 6.4 MB |

**Table 5.1:** The favourable workload for PIM-Rec for which we observed 9.61X speedup

Although the workload used in these experiments did not utilize the full memory capacity of each DPU, the achieved speedup was still significant. We proceeded to explore additional workloads to investigate how PIM-Rec's speedup compares to the baseline CPU implementation under different circumstances.

Next, we assessed the impact of varying the amount of embedding data stored on each DPU. We kept all other workload characteristics the same as the baseline (see Table 5.2) and only changed the size of the embedding data on each DPU. We change the size of data stored on each DPU by increasing the size of each embedding table. Figure 5.2 presents the results of this experiment.

As depicted in Figure 5.2, PIM-Rec scales well for larger embedding data sizes and can offer up to 10.5X inference speedup when 32 MB of embedding data is stored on each DPU. That would be a workload with 8 MB of embeddings per table. In other words, the workload for which we saw 10.5X speedup consists

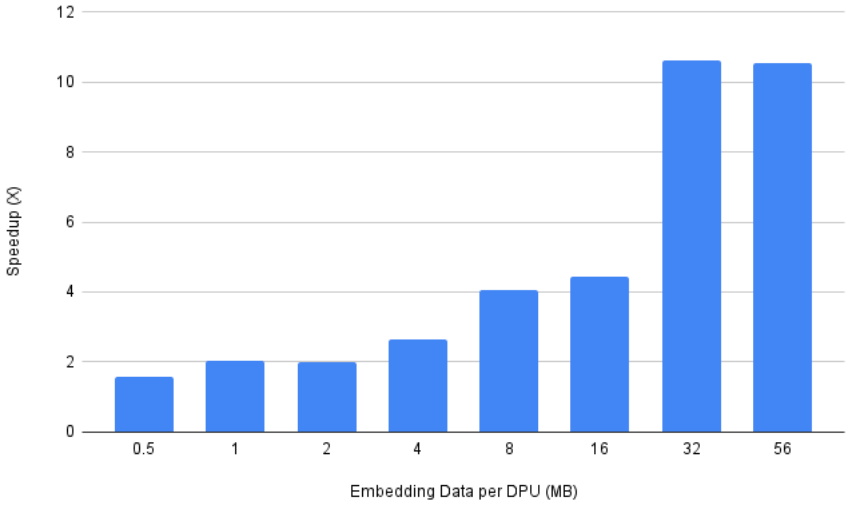



**Figure 5.2:** PIM-Rec inference cycle speedup compared to default CPU implementation when varying the size of embedding data stored on each DPU - table 5.2 shows complete setup

| # DPUs | query size | # emb tables | emb table size |
|---|---|---|---|
| 2048 | 30.72 KB | 32 | 1.024 to 114.688 GB |

**Table 5.2:** Baseline workload characteristics for experiments in figure 5.2

of embedding tables that have 8 MB rows each. Considering each table has 64 columns, this represents a medium-sized embedding table of size 0.5 GB. Even for smaller embedding tables, PIM-Rec still provides a promising speedup of around 2X, when all other characteristics are maintained as per Table 5.2.

Subsequently, we explored how PIM-Rec's speedup changes when varying the number of DPUs used. The number of DPUs is determined based on our design discussed in chapter 4, where each DPU stores one column of one table. Therefore, the total number of DPUs used by PIM-Rec is equal to $nr_column * nr_tables$. So there are 2 ways to change the number of DPUs being used, either by changing the number of embedding tables or changing the number of columns per table. Here we have decided to change the number of embedding tables since it results in a

model architecture closer to real worklaods. The baseline workload characteristics for this experiment are shown in Table 5.3, and the results are presented in Figure 5.3.

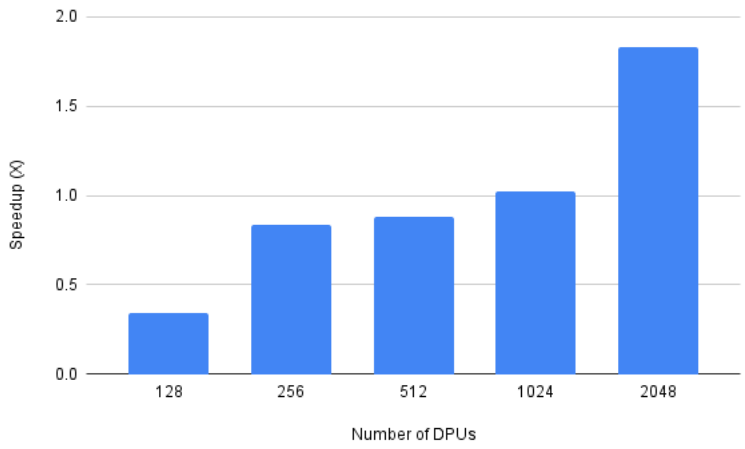



**Figure 5.3:** PIM-Rec inference cycle speedup compared to default CPU implementation when varying the number of embedding tables workload has - table 5.3 shows complete setup

| # DPUs | query size | emb data per DPU | emb table size |
|---|---|---|---|
| 128 to 2048 | 30.72 KB | 2 MB | 128 MB |

**Table 5.3:** Baseline workload characteristics for experiments in figure 5.3

Looking at table 5.3 we can see the workloads used for figure 5.3 are in the range of the smaller workloads seen in figure 5.2 because they are storing small amounts of embedding data per DPU as well as fewer number of DPUs. Hence they also offer smaller speedups. So far what we learned is that the speedup for smaller workloads is smaller and this trend can be seen in both figures 5.2 and 5.3.

Additionally, we investigated how PIM-Rec's speedup is affected by changing the size of the query processed within each inference cycle, which, in turn, modifies the amount of query data transferred from the host to DPUs and the size of the lookup results copied back from the DPUs to the host. This was achieved by vary-

ing the batch size. As we know each lookup query is a batch of lookup operations. By changing the batch size we can change both the size of query being copied to DPUs and the size of embedding lookup results. The baseline workload for this experiment is shown in table 5.4 and the experiment results can be seen in figure 5.4.

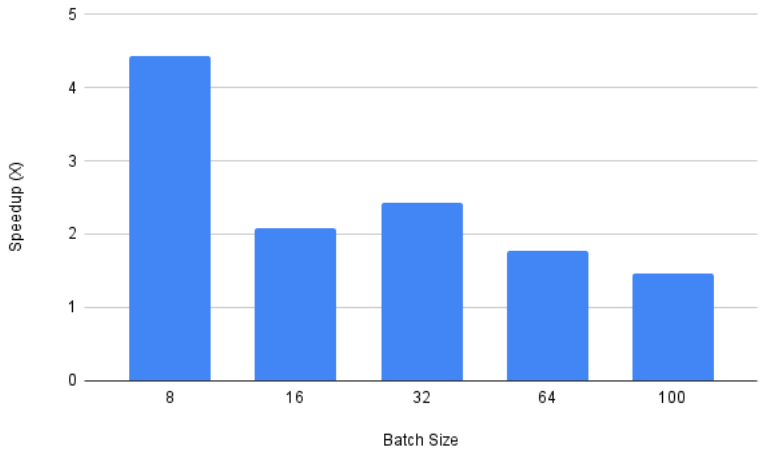



**Figure 5.4:** PIM-Rec inference cycle speedup compared to default CPU implementation when varying the the batch size - table 5.4 shows complete setup

| # DPUs | query size | emb data per DPU | emb table size |
|---|---|---|---|
| 2048 | 3.84 to 48 KB | 2 MB | 128 MB |

**Table 5.4:** Baseline workload characteristics for experiments in figure 5.4

From the results presented in Figure 5.4, we observed that PIM-Rec does not scale well for large batch sizes, and smaller batches can offer better speedup. Several factors could contribute to this behavior, such as the possibility of spending more time in DPUs when processing a larger batch or encountering higher latency in memory transfers between the host and DPUs. The increased size of batches affects both the data processing within the DPUs and the data transfer between the host and DPUs during an inference cycle. In Section 5.4, we will conduct a latency

breakdown analysis to gain insights into the reasons behind the observed speedup variations for different batch sizes in PIM-Rec.

## 5.2 Processor Metrics: IPC

Our expectation is that offloading memory-intensive lookups, which have low compute intensity, will free up resources for more compute-intensive operations to be performed by the CPU. As mentioned in Section 3, using the CPU for embedding lookups resulted in a low Instructions Per Cycle (IPC) for the host CPU. Thus, we anticipate that offloading lookups to DPUs will lead to a higher IPC for the host CPU. This increase can be attributed to two factors: First, the CPU is no longer burdened with memory-intensive, low-compute lookups, and second, the CPU will primarily focus on processing the more compute-intensive MLP layers of the network.

To test this hypothesis, we monitored the IPC for both the CPU and DPU and compared the measured IPC for the same workload between CPU and DPU lookups. It is worth noting that we monitored IPC for the entire inference cycle, not solely for the embedding lookup phase. To ensure greater accuracy, we conducted 100 measurements for each workload and considered their average values. The results of these experiments are illustrated in Figure 5.5. For a comprehensive overview of the workload characteristics in these experiments, please refer to Table 5.5. The experiments were conducted using a system equipped with an Intel Xeon Silver 4214 CPU.

| # DPUs | query size | emb data per DPU | emb data size | batch size |
|---|---|---|---|---|
| 2048 | 30.72 KB | 0.5 to 8 MB | 1 to 16 GB | 64 |

**Table 5.5:** Workload characteristics for experiments in figure 5.5

Key observations from Figure 5.5 are as follows:

- DPU IPC consistently remains at one. Given that the DPU has a single core, an IPC of 1 indicates that the processor is being fully utilized. PIM-Rec effectively maintains a full pipeline on the DPU by utilizing multiple tasklets for processing the lookups. In this experiment, 14 tasklets were used.

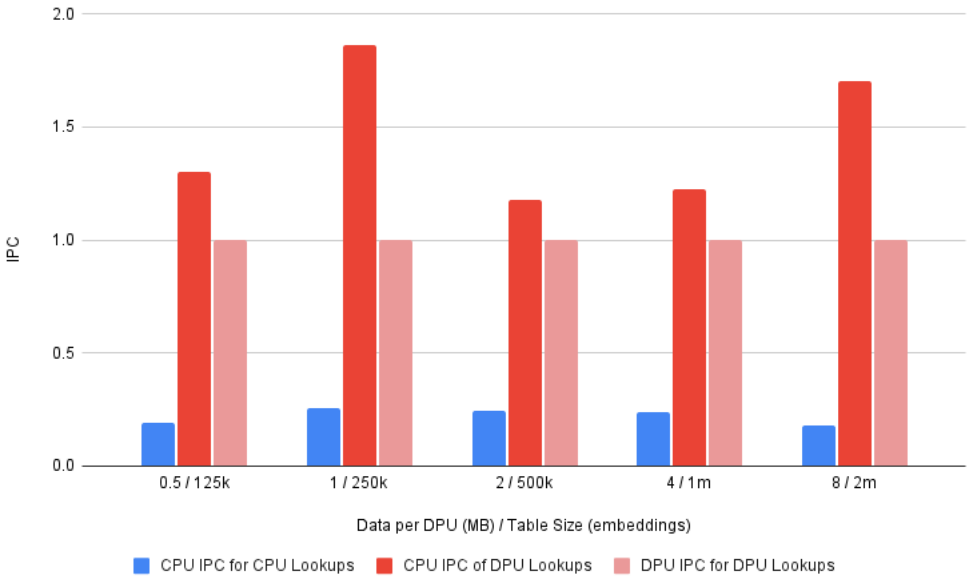



**Figure 5.5:** CPU and DPU IPC within an inference cycle for workload shown in table 5.5 with a variable amount of embedding data per DPU

- In all cases, when lookups are offloaded to DPUs, CPU IPC shows a significant increase compared to the default CPU lookup. This confirms our hypothesis that offloading lookups to PIM-Rec enhances CPU efficiency, enabling the main processor to focus on more compute-intensive tasks.
- The Intel Xeon Silver 4214 CPU used in these experiments has 16 cores. Despite the considerable improvement in IPC achieved by PIM-Rec, the observed IPC is still below the maximum potential of this CPU. This suggests that there may be further room for improvements in CPU utilization.
- There is no clear trend in IPC as we scale the workload and store more embedding data on DPUs. To gain more insights into IPC scaling, we should consider other workload parameters.

To explore how IPC scales with varying workloads, we conducted another experiment by altering the number of DPUs used. This time, all workload characteristics, including the amount of embedding data stored on each DPU, remained the

same, while the number of embedding tables in the workload was changed. This variation directly impacted the number of DPUs used for storing the tables. The results of these experiments are depicted in Figure 5.6, and the detailed workload characteristics are presented in Table 5.6.

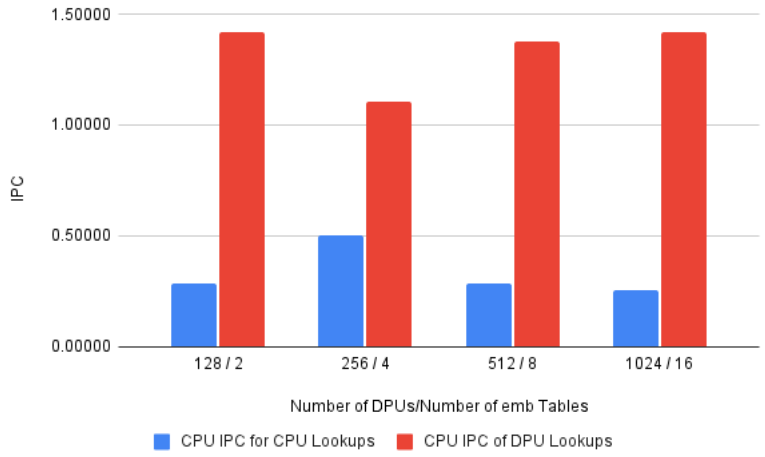



**Figure 5.6:** CPU IPC within an inference cycle for workload shown in table 5.6 with variable number of DPUs used for storing embedding data

| # DPUs | query size | emb data per DPU | emb data size | batch size |
|---|---|---|---|---|
| 128 to 2048 | 30.72 KB | 2 MB | 256 MB to 4 GB | 64 |

**Table 5.6:** Workload characteristics for experiments in figure 5.6

As the DPU IPC consistently remained at 1 in our experiments, we have omitted showing the DPU IPC in Figure 5.6 to focus solely on the comparison of CPU IPC. Once again, we observe that CPU IPC is significantly higher for DPU implementation compared to the CPU one. However, no specific trend in IPC is evident while scaling the workload in this experiment as well.

To further investigate scalability, we explored another aspect by varying the batch size, thereby altering the amount of data transferred to and from the DPUs in each inference cycle. We monitored the IPC and observed how it changed accordingly. The results of this experiment are displayed in Figure 5.7, and the workload

characteristics are detailed in Table 5.7.

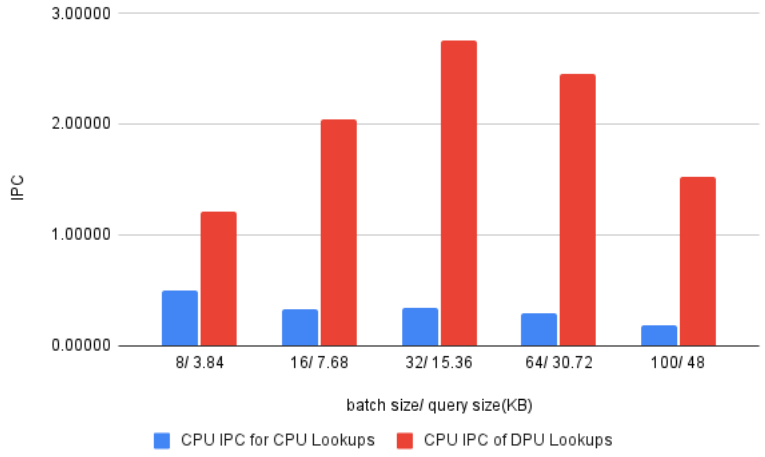



**Figure 5.7:** CPU IPC within an inference cycle for workload shown in table 5.7 with variable batch size

| # DPUs | query size | emb data per DPU | emb data size | batch size |
|---|---|---|---|---|
| 2048 | 3.84 to 48 KB | 2 MB | 4 GB | 8 to 100 |

**Table 5.7:** Workload characteristics for experiments in figure 5.7

Indeed, the results consistently demonstrate that CPU IPC is significantly higher when lookups are offloaded to PIM-Rec, affirming the enhanced CPU efficiency achieved through offloading. Although no specific trend in CPU IPC emerges when the batch size and query size are altered, our primary objective of achieving better CPU efficiency with PIM-Rec, compared to the baseline, has been successfully met. The offloading of lookups to PIM-Rec has effectively improved the utilization of the main processor, making it more capable of handling compute-intensive tasks.

## 5.3 Cache Metrics: Hit Rate

The unpredictable memory access pattern of embedding lookups, with data accessed over a large working set and accessed locations being highly random, results in a low cache hit rate for DLRM inference, as observed in Section 3. We anticipate that PIM-Rec will mitigate this issue to some extent. By storing embedding data in PIM modules and removing the CPU's involvement in lookups, we expect a reduction in cache misses. In PIM-Rec, the CPU primarily handles the MLP layers of the network, which are known to cause fewer cache misses. To verify if this expectation holds true for PIM-Rec, we conducted experiments illustrated in Figure 5.8. These experiments examine the cache hit rates and explore how PIM-Rec affects cache behavior.

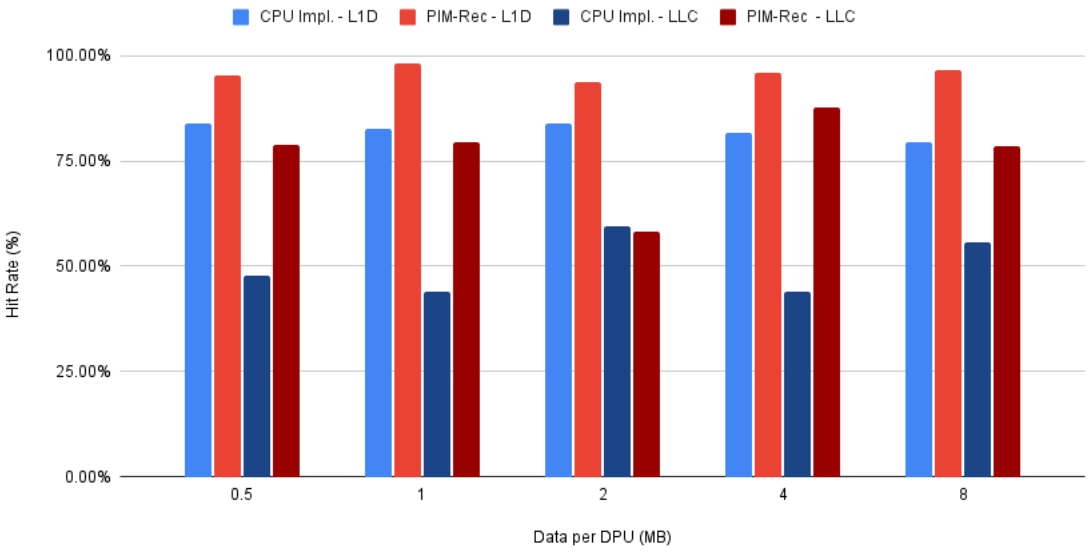



**Figure 5.8:** cache hit rate within an inference cycle for workload shown in table 5.8 with variable amount of embedding data stored per DPU

From the experiment results shown in Figure 5.8, we can draw the following conclusions:

- For both LLC and L1D, the cache hit rate for PIM-Rec is higher than the baseline CPU implementation in almost all cases.

| # DPUs | query size | emb data per DPU | emb data size | batch size |
|---|---|---|---|---|
| 2048 | 30.72 KB | 0.5 to 8 MB | 1 to 16 GB | 64 |

**Table 5.8:** Workload characteristics for experiments in figure 5.8

- The increase in LLC cache hit rate is particularly significant. When 4 MB of embedding data per DPU is used, PIM-Rec's LLC cache hit rate is nearly double that of the CPU baseline implementation. The difference in LLC hit rate is close to zero only for 2 MB of embedding data per DPU.
- The L1D hit rate for PIM-Rec is consistently higher with no exceptions.
- There is no specific trend in cache hit rate as we scale the workload.

Furthermore, we examined the cache hit rate when varying the number of DPUs used. To achieve this, we changed the number of embedding tables in PIM-Rec, which impacted the number of DPUs utilized. The experiment setup is detailed in Table 5.9, and the results are shown in Figure 5.9.

| # DPUs | query size | emb data per DPU | emb data size | batch size |
|---|---|---|---|---|
| 128 to 2048 | 30.72 KB | 2 MB | 256 MB to 4 GB | 64 |

**Table 5.9:** Workload characteristics for experiments in figure 5.9

In Figure 5.9, we observe that PIM-Rec consistently maintains a higher LLC hit rate compared to the CPU baseline, and this gap between PIM-Rec's LLC hit rate and the baseline increases as we use more DPUs. The reason behind this phenomenon is that the CPU's LLC hit rate decreases with a larger number of tables, while PIM-Rec's LLC hit rate increases. Although the L1D hit rate for both implementations is relatively similar, PIM-Rec performs slightly better in all cases.

In conclusion, PIM-Rec has successfully achieved the intended goal in terms of cache hit rate. Throughout all the experiments we conducted, PIM-Rec consistently exhibits a higher cache hit rate compared to its CPU baseline implementation. This demonstrates that by offloading lookups to PIM-Rec and allowing the

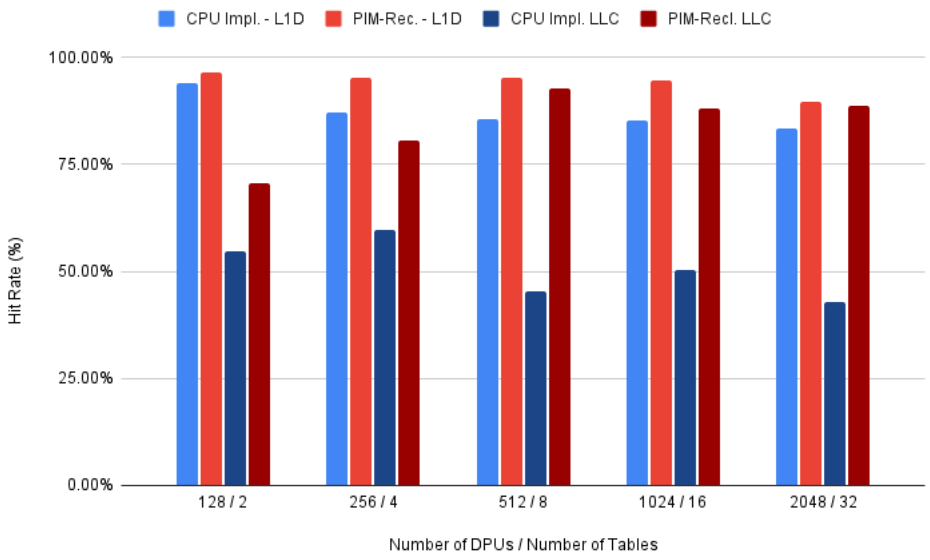



**Figure 5.9:** Cache LLC and L1D hit rate within an inference cycle for workload shown in table 5.9 with variable number of DPUs used for storing embedding data

CPU to primarily handle the compute-intensive MLP layers, PIM-Rec effectively reduces cache misses and improves cache efficiency.

## 5.4 Embedding Layer Latency Breakdown

Understanding the latency breakdown of PIM-Rec's embedding lookups is crucial in identifying the stages of the execution pipeline that scale well and those that may present challenges in terms of scalability. This breakdown helps us determine which workloads can benefit the most from switching to PIM-Rec by leveraging the stages that scale effectively while minimizing stress on less scalable stages.

PIM-Rec's lookup execution pipeline consists of four main stages:

1. **Query Copy to DPUs:** In this stage, when a query is received from DLRM, it is copied from the host to the DPUs and stored in MRAM. The host efficiently distributes the relevant parts of the query to each DPU, ensuring that each DPU receives the query segments meant for the specific part of the embedding data it

stores. Memory transfers to DPUs are done in bulk, allowing for parallel transfers across all DPUs.

2. **Processing in DPU:** Once DPUs receive the query and understand their respective embedding lookup tasks, they begin executing and processing the query to generate the lookup results. The results are then stored in MRAM for later access by the host.

3. **Result Copy from DPU to Host:** After DPUs complete the processing, they inform the host, which then proceeds to copy the lookup results from MRAM back to the main memory of the system. This memory transfer is asynchronous, accommodating variations in processing times among different DPUs.

4. **Query Post-processing:** Once all the embedding results from a DPU are copied back to the host, post-processing begins. This stage involves converting the 32-bit integers back to 32-bit floating-point values. While DPUs process data as 32-bit integers due to a lack of floating-point operators, the original format of the data was floating-point. The post-processing task leverages the host's multithreading capability to handle queries from different embedding tables in parallel using different threads. Similar to the previous stage, this process is also asynchronous, considering the asynchronous nature of the previous stage, the result copy, which may finish at different times for different DPUs.

Having identified the stages of the lookup pipeline, we can now examine some experimental results. Figure 5.10 illustrates the lookup latency breakdown as we scale the number of DPUs used, achieved by altering the number of embedding tables. A comprehensive overview of the workload characteristics for these experiments can be found in Table 5.10. These results will provide insights into how each stage of the PIM-Rec pipeline contributes to the overall latency and how the pipeline behaves with varying numbers of DPUs.

| # DPUs | query size | emb data per DPU | emb data size | batch size |
|---|---|---|---|---|
| 128 to 2048 | 30.72 KB | 2 MB | 256 MB to 4 GB | 64 |

**Table 5.10:** Workload characteristics for experiments in figure 5.10

The latency breakdown depicted in Figure 5.10 provides valuable insights into the behavior of PIM-Rec's lookup pipeline for different workloads involving vary-

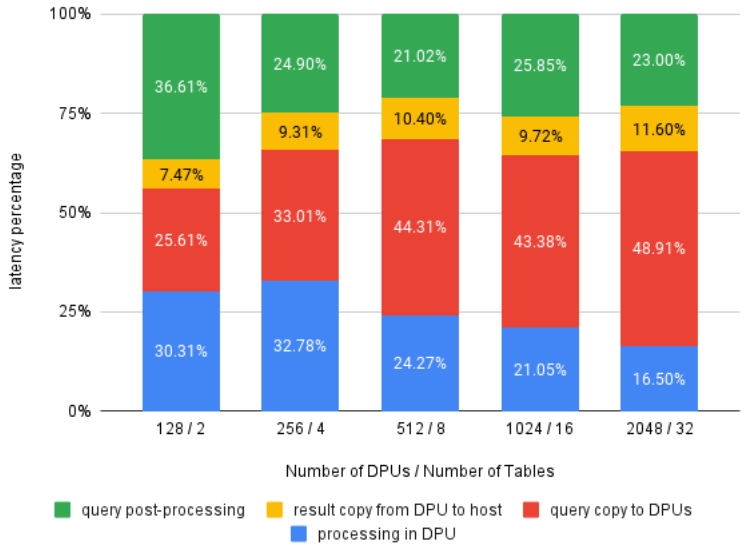



**Figure 5.10:** PIM-Rec Lookup latency breakdown for workload shown in table 5.10 with variable number of DPUs used for storing embedding data

ing numbers of DPUs or embedding tables:

1. For workloads with a smaller number of DPUs or embedding tables, the lookup latency in PIM-Rec is more evenly distributed between the DPU processing, post-processing, and query copying stages. The share of query copy to DPUs remains relatively small compared to the other stages. 2. However, as the number of embedding tables increases and more DPUs are involved, the share of query copy to DPUs becomes more significant. This happens because the process of copying the correct queries to numerous DPUs requires more time, leading to a larger portion of the overall latency being attributed to this stage. 3. The share of DPU processing remains relatively constant since the time spent on processing within each DPU remains consistent regardless of the number of DPUs involved. 4. Similar to DPU processing, query post-processing exhibits a fairly constant latency, and its share of the lookup latency decreases as the number of DPUs increases. 5. The result of copying data back to the host takes up a small portion of the overall latency in all cases. However, its share increases when more DPUs

are involved due to the additional time required for memory transfer from multiple DPUs.

To further investigate the impact of the amount of embedding data stored by each DPU on the latency breakdown, we conducted experiments illustrated in Figure 5.11. In these experiments, we kept the number of DPUs constant and varied the amount of embedding data stored by each DPU, ranging from 0.5 MB to the maximum capacity of 56 MB. For a comprehensive description of the experimental setup, please refer to Table 5.11. These experiments provide insights into how changes in the size of embedding data affect each stage of the PIM-Rec lookup pipeline and its overall latency distribution.

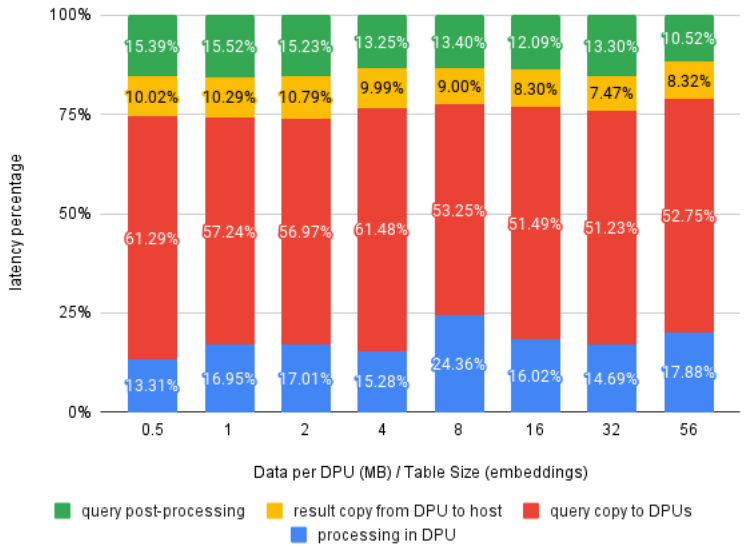



**Figure 5.11:** PIM-Rec Lookup latency breakdown for workload shown in table 5.11 with variable amount of embedding data stored per DPU

| # DPUs | query size | emb data per DPU | emb data size | batch size |
|---|---|---|---|---|
| 2048 | 30.72 KB | 0.5 to 56 MB | 1 to 114 GB | 64 |

**Table 5.11:** Workload characteristics for experiments in figure 5.11

In Figure 5.11, we observe that storing more data per DPU does not signifi-

cantly affect the latency caused by processing within the DPUs. Although there are some fluctuations, there is no clear trend, and the share of latency breakdown remains relatively constant. While the absolute value of DPU processing latency increases as more data is stored, it does not escalate as rapidly as the latency for query copying. The majority of PIM-Rec's lookup latency is attributed to the process of copying queries to DPUs.

As previously seen, the latency caused by copying results back from DPUs to the host is minimal, as the size of the result data being transferred is considerably smaller than the query data.

Given the substantial impact of query copy in PIM-Rec's lookup latency, it is intriguing to investigate how the breakdown changes when altering the size of the queries. By modifying the batch size, we can change both the query size being copied to DPUs and the result size being copied back to the host. The results of these experiments are depicted in Figure 5.12, and the workload characteristics can be found in Table 5.12. These experiments shed light on how changes in query size affect each stage of the PIM-Rec lookup pipeline and its overall latency distribution.

| # DPUs | query size | emb data per DPU | emb data size | batch size |
|---|---|---|---|---|
| 2048 | 3.84 to 48 KB | 2 MB | 4 GB | 8 to 100 |

**Table 5.12:** Workload characteristics for experiments in figure 5.12

The findings in Figure 5.12 are indeed reassuring. Despite the introduction of larger queries with larger batch sizes, the query copying latency does not exhibit a significant increase in its share of the latency breakdown. This indicates that PIM-Rec is capable of handling larger batch sizes without suffering a massive increase in latency caused by data transfer.

Overall, the results from PIM-Rec's latency breakdown align well with our expectations. Throughout various scaling scenarios, no red flags were observed in any stages of PIM-Rec's latency breakdown. This provides us with confidence that PIM-Rec is well-equipped to handle larger lookup workloads without compromising on efficiency and latency. The scalability demonstrated by PIM-Rec in different directions further validates its potential as an efficient solution for handling a wide

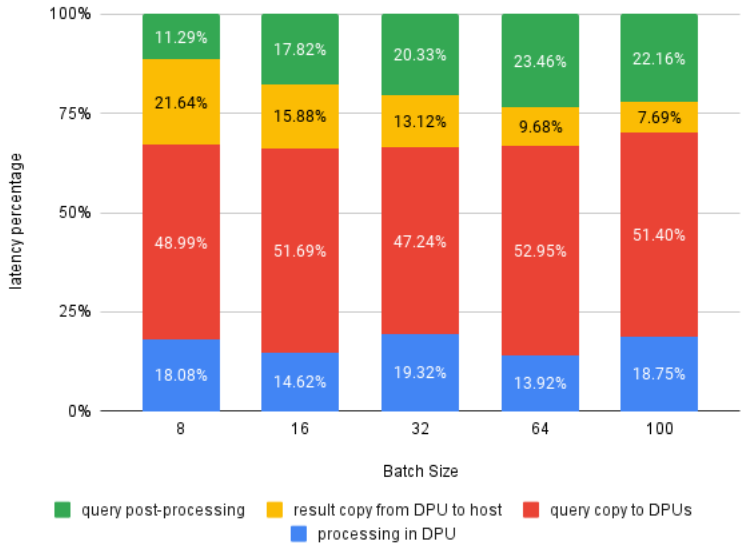



**Figure 5.12:** PIM-Rec lookup latency breakdown for workload shown in table 5.12 with variable batch size

range of workloads.

# Chapter 6

# Conclusion

The experimentation with PIM-Rec revealed notable advantages in terms of reducing latency and increasing throughput for inference on Recommender Models. Employing PIM technology holds promise, but it is essential to approach the system's design pragmatically, considering that implementation overhead could potentially overshadow its speedup benefits. Although finely specialized hardware designs may seem attractive, they are often impractical and challenging to justify for deployment in rapidly changing large datacenters. A more feasible approach lies in adopting a general-purpose PIM solution, even if it may not yield miraculous performance enhancements. Nonetheless, given the immense scale of inference workloads at hyperscalar datacenters, even slight improvements in latency and throughput can translate to substantial cost savings, potentially amounting to millions of dollars.

The key aspect of our scalable design is the correlation between computing resources and memory size. As the memory size increases, the system accommodates a proportional increase in processors. This approach ensures scalability and flexibility, crucial factors for hyper scalars, both of which are addressed effectively by PIM-Rec.

## 6.1 Future Work

PIM-Rec still has the potential to incorporate more complex optimizations. Currently, we are actively engaged in testing PIM-Rec with DLRM instances trained on real datasets. Collaboration with researchers from ETH Zurich is underway to evaluate PIM-Rec's performance on recommendation models included in the MLPerf[18] inference benchmark suite. We expect to observe similar results and speedups to those demonstrated in our experiments with randomly generated workloads. Furthermore, we anticipate leveraging the skewed key access pattern present in real datasets to further optimize PIM-Rec lookups.

However, the servers currently at our disposal can only accommodate one of the benchmarks provided by MLPerf due to two limiting factors: the size of the dataset used for model training and the model size itself. The dataset's size presents a constraint because the MLPerf implementation provided by the organization requires users to load and pre-process the entire dataset, necessitating a machine with almost 256 GB of memory for a 100 GB dataset. The Criteo Kaggle dataset[3] is 12 GB, which our current servers can handle, but the Terabyte dataset at 91 GB[2] exceeds their capacity. Therefore, the only MLPerf benchmark we can currently manage with our servers is the DLRM instance trained on the Criteo Kaggle dataset[3], which has a memory footprint of 1 GB for the entire model, including both the embedding and MLP layers. Fortunately, the embedding data fits comfortably within our server's UPMEM DIMMs, and the dataset's size permits straightforward loading and pre-processing on our server.

Regrettably, the other two benchmarks are beyond our current server capabilities:

- The DLRM instance trained on the Terabyte dataset with a model size of 10 GB can readily fit in memory, with our server's UPMEM DIMMs able to handle the embedding data and the conventional memory accommodating the MLP layers. However, loading and pre-processing the dataset pose challenges.

- The DLRM instance trained again on the Terabyte dataset with a model size of 100 GB faces constraints in both storing the model and pre-processing the dataset. Loading the entire model into conventional memory at the outset

presents issues with Python's memory management and garbage collector, as it takes time to clear out embedding data that is no longer accessed within the conventional memory.

To overcome these limitations, we are actively pursuing access to servers equipped with UPMEM DIMMs and larger conventional memory sizes, which would allow us to conduct experiments on all the MLPerf benchmarks. In the meantime, we are also exploring alternative approaches to dataset pre-processing, such as trying pre-processing on another server and then transferring the pre-processed files to the PIM-equipped server. Additionally, we are modifying sections of the code responsible for dataset loading and pre-processing.

Overall, we are confident that with these efforts, we will soon be able to evaluate PIM-Rec's capabilities with all three models, providing valuable insights into its performance.

# Bibliography


[1] Meta training and inference accelerator (ai mtia). https://ai.meta.com/blog/meta-training-inference-accelerator-AI-MTIA/, 2023. Accessed: July 22, 2023. → page 16

[2] Download terabyte click logs dataset on criteo. https://labs.criteo.com/2013/12/download-terabyte-click-logs/, 2023. Accessed on 17/07/2023. → pages 18, 19, 53

[3] Kaggle display advertising challenge dataset on criteo. https://labs.criteo.com/2014/02/kaggle-display-advertising-challenge-dataset/, 2023. Accessed on 17/07/2023. → pages ix, x, 18, 20, 21, 22, 23, 53

[4] M. Adnan, Y. E. Maboud, D. Mahajan, and P. J. Nair. Accelerating recommendation system training by leveraging popular choices. *Proc. VLDB Endow.*, 15(1):127–140, sep 2021. ISSN 2150-8097. doi:10.14778/3485450.3485462. URL https://doi.org/10.14778/3485450.3485462. → page 14

[5] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 105–117, 2015. doi:10.1145/2749469.2750386. → page 3

[6] H.-T. Cheng, L. Koc, J. Harmsen, T. Shaked, T. Chandra, H. Aradhye, G. Anderson, G. Corrado, W. Chai, M. Ispir, R. Anil, Z. Haque, L. Hong, V. Jain, X. Liu, and H. Shah. Wide deep learning for recommender systems, 2016. → page 1

[7] M. N. et al. An implementation of deep learning recommendation model for personalization and recommendation systems. https://github.com/facebookresearch/dlrm, 2019. → page 2

[8] A. Firoozshahian, J. Coburn, R. Levenstein, R. Nattoji, A. Kamath, O. Wu, G. Grewal, H. Aepala, B. Jakka, B. Dreyer, A. Hutchin, U. Diril, K. Nair, E. K. Aredestani, M. Schatz, Y. Hao, R. Komuravelli, K. Ho, S. Abu Asal, J. Shajrawi, K. Quinn, N. Sreedhara, P. Kansal, W. Wei, D. Jayaraman, L. Cheng, P. Chopda, E. Wang, A. Bikumandla, A. Karthik Sengottuvel, K. Thottempudi, A. Narasimha, B. Dodds, C. Gao, J. Zhang, M. Al-Sanabani, A. Zehtabioskuie, J. Fix, H. Yu, R. Li, K. Gondkar, J. Montgomery, M. Tsai, S. Dwarakapuram, S. Desai, N. Avidan, P. Ramani, K. Narayanan, A. Mathews, S. Gopal, M. Naumov, V. Rao, K. Noru, H. Reddy, P. Venkatapuram, and A. Bjorlin. Mtia: First generation silicon targeting meta's recommendation systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700958. doi:10.1145/3579371.3589348. URL https://doi.org/10.1145/3579371.3589348. → pages 16, 17, 25

[9] S. Ghose, A. Boroumand, J. S. Kim, J. Gómez-Luna, and O. Mutlu. Processing-in-memory: A workload-driven perspective. *IBM Journal of Research and Development*, 63(6):3:1–3:19, 2019. doi:10.1147/JRD.2019.2934048. → page 3

[10] U. Gupta, X. Wang, M. Naumov, C. Wu, B. Reagen, D. Brooks, B. Cottel, K. M. Hazelwood, B. Jia, H. S. Lee, A. Malevich, D. Mudigere, M. Smelyanskiy, L. Xiong, and X. Zhang. The architectural implications of facebook's dnn-based personalized recommendation. *CoRR*, abs/1906.03109, 2019. URL https://arxiv.org/abs/1906.03109. → pages ix, 2, 10, 13, 14

[11] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 982–995, 2020. doi:10.1109/ISCA45697.2020.00084. → page 11

[12] C. Huyen. *Designing Machine Learning Systems: An Iterative Process for Production-ready Applications*. O'Reilly Media, Incorporated, 2022. ISBN 9781098107963. URL https://books.google.ca/books?id=BAy_zgEACAAJ. → page 25

[13] R. Hwang, T. Kim, Y. Kwon, and M. Rhu. Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer*

*Architecture*, ISCA '20, page 968–981. IEEE Press, 2020. ISBN 9781728146614. doi:10.1109/ISCA45697.2020.00083. URL https://doi.org/10.1109/ISCA45697.2020.00083. → page 15

[14] L. Ke, U. Gupta, C.-J. Wu, B. Y. Cho, M. Hempstead, B. Reagen, X. Zhang, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, and X. Wang. Recnmp: Accelerating personalized recommendation with near-memory processing, 2019. → pages 2, 3, 14

[15] L. Ke, U. Gupta, B. Y. Cho, D. Brooks, V. Chandra, U. Diril, A. Firoozshahian, K. Hazelwood, B. Jia, H.-H. S. Lee, M. Li, B. Maher, D. Mudigere, M. Naumov, M. Schatz, M. Smelyanskiy, X. Wang, B. Reagen, C.-J. Wu, M. Hempstead, and X. Zhang. Recnmp: Accelerating personalized recommendation with near-memory processing. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 790–803. IEEE Press, 2020. ISBN 9781728146614. doi:10.1109/ISCA45697.2020.00070. URL https://doi.org/10.1109/ISCA45697.2020.00070. → page 14

[16] Y. Kwon, Y. Lee, and M. Rhu. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 740–753, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369381. doi:10.1145/3352460.3358284. URL https://doi.org/10.1145/3352460.3358284. → page 15

[17] Y.-C. Kwon, S. H. Lee, J. Lee, S.-H. Kwon, J. M. Ryu, J.-P. Son, O. Seongil, H.-S. Yu, H. Lee, S. Y. Kim, Y. Cho, J. G. Kim, J. Choi, H.-S. Shin, J. Kim, B. Phuah, H. Kim, M. J. Song, A. Choi, D. Kim, S. Kim, E.-B. Kim, D. Wang, S. Kang, Y. Ro, S. Seo, J. Song, J. Youn, K. Sohn, and N. S. Kim. 25.4 a 20nm 6gb function-in-memory dram, based on hbm2 with a 1.2tflops programmable computing unit using bank-level parallelism, for machine learning applications. In *2021 IEEE International Solid- State Circuits Conference (ISSCC)*, volume 64, pages 350–352, 2021. doi:10.1109/ISSCC42613.2021.9365862. → page 3

[18] MLCommons. MLCommons/Inference: Recommendation - DLRM PyTorch. https://github.com/mlcommons/inference/tree/master/recommendation/dlrm/pytorch, Accessed 2023-07-18. → pages ix, x, 18, 19, 20, 21, 22, 23, 53

[19] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini, D. Dzhulgakov, A. Mallevich, I. Cherniavskii, Y. Lu, R. Krishnamoorthi, A. Yu, V. Kondratenko, S. Pereira, X. Chen, W. Chen, V. Rao, B. Jia, L. Xiong, and M. Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems, 2019. → pages ix, 2, 10, 12, 13, 18

[20] J. W. Niloofar Zarif. Pim-rec fork of dlrm. https://github.com/UBC-ECE-Sasha/PIM-dlrm/tree/bb593a4a6d68de6d3257ac3e4fccdf26e929a1cc, 2022. Accessed: July 22, 2023. → page 34

[21] J. W. Niloofar Zarif. Pim-rec on github. https://github.com/UBC-ECE-Sasha/PIM-Embedding-Lookup, 2022. Accessed: July 22, 2023. → page 34

[22] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou. Mlperf inference benchmark. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 446–459, 2020. doi:10.1109/ISCA45697.2020.00045. → page 18

[23] U. SAS. UPMEM Company upmem organization, 2023. URL https://www.upmem.com/company/. → page 2

[24] U. SAS. UPMEM PIM Technology upmem website, 2023. URL https://www.upmem.com/technology/. → pages x, 2, 3, 28

[25] A. Saulsbury, F. Pong, and A. Nowatzyk. Missing the memory wall: The case for processor/memory integration. *SIGARCH Comput. Archit. News*, 24 (2):90–101, may 1996. ISSN 0163-5964. doi:10.1145/232974.232984. URL https://doi.org/10.1145/232974.232984. → page 2

[26] Wikipedia contributors. Random access memory and memory wall, 2023. URL https://en.wikipedia.org/wiki/Random-access_memory#Memory_wall. [Online; accessed 09-July-2023]. → page 2