



SEMS: Scalable Embedding Memory System for Accelerating Embedding-Based DNNs

Sejin Kim , Jungwoo Kim , Yongjoo Jang ,
Jaeha Kung , and Sungjin Lee 

Abstract—Embedding layers, which are widely used in various deep learning (DL) applications, are very large in size and are increasing. We propose scalable embedding memory system (SEMS) to deal with the inference of DL applications with a large embedding layer. SEMS is built using scalable embedding memory (SEM) modules, which include FPGA for acceleration. In SEMS, PCIe bus, which is scalable and versatile, is used to expand the system memory and processing in SEMs reduces the amount of data transferred from SEMs to host, improving the effective bandwidth of PCIe. In order to achieve better performance, we apply various optimization techniques at different levels. We develop SEMlib, a Python library to provide convenience in using SEMS. We implement a proof-of-concept prototype of SEMS and using SEMS yields DLRM execution time that is $32.85\times$ faster than that of a CPU-based system when there is a lack of DRAM to hold the entire embedding layer.

Index Terms—DNN accelerators, embeddings, recommender systems, system for machine learning

1 INTRODUCTION

Embedding layers are widely employed in various deep learning (DL) applications, such as recommender systems (RSes), language modeling, and machine translation. These applications use large embedding layers for accuracy, occupying hundreds of GBs of memory [1]. The size of the embedding layers increases faster than the density of DRAM, which pushes us to populate more DRAM modules in the system. Unfortunately, owing to the limited number of DIMM slots, it becomes difficult to run DL applications requiring huge embedding layers.

In this paper, we propose a scalable embedding memory system, called *SEMS*, which expands the system memory for embedding layers in a cost-effective yet high performance manner. SEMS uses a commodity PCIe interconnect to expand the system memory. It is composed of multiple scalable embedding memory (SEM) modules, each of which employs a large DRAM (e.g., 64GB–256GB) and is attached to PCIe Gen3 \times 16. The PCIe bus has a scalable design and allows a large number of devices to be connected to the CPUs. Through PCIe switches that expand the number of endpoints, we can attach a theoretically unlimited number of SEMs, creating a huge memory pool.

SEMS is scalable in terms of capacity, but the limited bandwidth of the PCIe may act as a bottleneck, slowing down the applications. We address this issue by processing data using the FPGAs inside SEMs. SEMs keep the entire embedding layer in their internal DRAM and then perform reduce operations which are relatively simple to be accelerated by FPGAs. By sending only the reduced results to the host, SEMS improves the effective bandwidth of PCIe

• The authors are with the Department of Electrical Engineering & Computer Science, DGIST, Daegu 42988, South Korea. E-mail: {sejink06, jungwoo, dracol, jhkung, sungjin.lee}@dgist.ac.kr.

Manuscript received 2 November 2022; accepted 23 November 2022. Date of current version 22 December 2022.

This work was supported in part by the MOTIE (Ministry of Trade, Industry & Energy) under Grant 1415181081, in part by KSRC (Korea Semiconductor Research Consortium) under Grant 20019402, in part by development of the future semiconductor device, and in part the by the Samsung Research Funding Incubation Center of Samsung Electronics under Grant SRFCIT1902-03. (Corresponding author: Sungjin Lee.)

Digital Object Identifier no. 10.1109/LCA.2022.3227560

by several thousand times. To efficiently perform the reduce operations we also optimize SEMS's FPGA accelerators using various techniques.

SEMS is designed to embrace embedding layers that are too large to fit in a single SEM. It partitions the embedding layer into pieces and assigns them over multiple SEM modules. Each piece has no computational dependency with other pieces. This enables us to evenly distribute pieces in a manner that balances computational load and memory utilization across SEM modules. Moreover, since no synchronization among SEM modules is necessary, SEMS provides performance improvement scalable to the number of SEMs. We implement a Python library, called *SEMlib*, which virtualizes the embedding layer and the associated operations through well-defined APIs. This lets developers to easily integrate SEMS with DL applications.

We implement a proof-of-concept prototype of SEMS. SEM is built using a commercially available FPGA+DRAM module, Xilinx's Alveo U200 with 64 GB DRAM and an UltraScale+ FPGA [2]. U200 belongs to a high-end product family, but there are cheaper alternatives (e.g., BittWare's 250-M2D [3]). As the benchmark, we use Facebook's deep learning recommendation model (DLRM) [4]. SEMS shows $1.36\times$ longer execution time compared to a typical CPU-based system with enough memory to accommodate the entire embedding layer. However, for models with a huge embedding layer which exceeds the size of the system memory, SEMS shows $32.85\times$ higher performance.

2 DEEP LEARNING-BASED RECOMMENDER SYSTEMS

Recommender Systems (RSes) recommend items to users by producing a click-through-rate for given inputs. Nowadays, DL-based RSes based on embeddings are widely deployed in various domains such as advertisements and content recommendations because of its high accuracy and memory efficiency.

The overall structure of typical DL-based RSes is shown in Fig. 1. The model consists of four main components: (i) a bottom MLP, (ii) an embedding layer, (iii) feature interaction, and (iv) a top MLP. The bottom MLP takes in dense features and produces a vector. The embedding layer takes in sparse features and outputs a vector per sparse feature. The outputs of the bottom MLP and the embedding layer, which are vectors of equal lengths, are the input of feature interaction. When the feature interaction operation is sum, a vector is fed into the top MLP. Finally, the top MLP produces a click-through rate.

Embedding Layer. Fig. 1 (right) illustrates an embedding layer that receives N_{spa} sparse features as its input. The embedding layer consists of N_{spa} embedding tables and each table T_i is used to process input sparse feature i . Each column of the embedding table is an embedding of an item of the corresponding sparse feature. The size of the embedding is called an embedding dimension (D).

The size of an embedding layer depends on (i) the number of embeddings in the embedding layer and (ii) D . For high accuracy, embedding layers tend to employ more tables [5]. In proportion to the increase in the number of items, the number of embeddings in each table increases. A high D is also preferred for high accuracy. As a result, embedding layers get larger, reaching hundreds of GBs in size [1].

Two operations are involved in an embedding layer: ① embedding lookup and ② aggregation. They are executed for each sparse feature. During embedding lookup (① in Fig. 1), the embedding(s) at the index indicated by the sparse feature is retrieved from the table. When each sparse feature requires $N_{lookups}$ embedding lookups, the $N_{lookups}$ embeddings are aggregated (② in Fig. 1) into a single vector through an element-wise sum. The number of the output

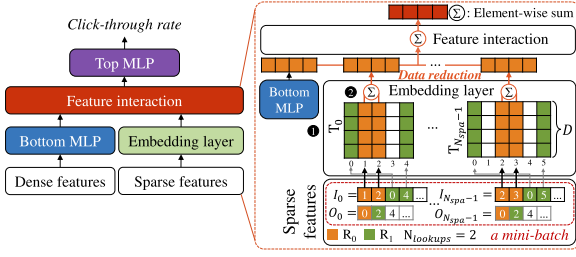


Fig. 1. The overall structure of the modern DL-based RSes.

vectors of the embedding layer is thus equal to the number of sparse features, N_{spa} .

To increase computational efficiency, the inference requests are grouped into a mini-batch and processed together. Sparse feature i for a mini-batch is represented as indices (I_i) and offsets (O_i), as shown in Fig. 1 (right). I_i is an array of indices of the embeddings to look up from T_i and O_i is an array of offsets that indicate the starting index of each request within I_i . Thus, for request R_j with request number j , embeddings at index $I_i[k]$ for k in range $[O_i[j], O_i[j+1])$ are looked up. For example, in Fig. 1 (right), for R_0 , from T_0 , the embeddings at indices 1 and 2 are looked up ($N_{lookups} = 2$). They are then aggregated into a single vector. While not shown in Fig. 1, the above operations are done for all the requests in the mini-batch. This produces $N_{spa} \cdot N_{req}$ vectors, where N_{req} is the number of inference requests in a mini-batch.

Feature Interaction. For each request, the $N_{spa} + 1$ vectors produced by the bottom MLP and the embedding layer are the input for feature interaction. Feature interaction performs arithmetic operations over the input vectors. Sum is equal to the aggregation operation of the embedding layer as it performs element-wise sum over the vectors, producing an output vector. It is the most popular as it reduces the size of data and is simple to implement. We thus use sum as the default operation.

3 DESIGN AND IMPLEMENTATION OF SEMS

3.1 Overall Architecture

Fig. 2 shows the overall design of SEMS that comprises multiple SEMs. Each SEM is connected to the PCIe and consists of an FPGA and an off-chip DRAM. The off-chip DRAM is used to store the entire embedding layer. Various operations, which were executed in CPUs or GPUs, are offloaded to the FPGA. The embedding tables and the offloaded operations are encapsulated by a well-defined Python library, SEMlib. As depicted in Fig. 2, SEMlib exposes two classes, *CETable* and *CETable*, which represent an embedding layer and an embedding table, respectively. A *CETable* instance is created to build an embedding layer over SEMS, which involves the creation of *CETable* instances. *CETables* are assigned to SEM modules in a way that maximally exploits SEMS-level

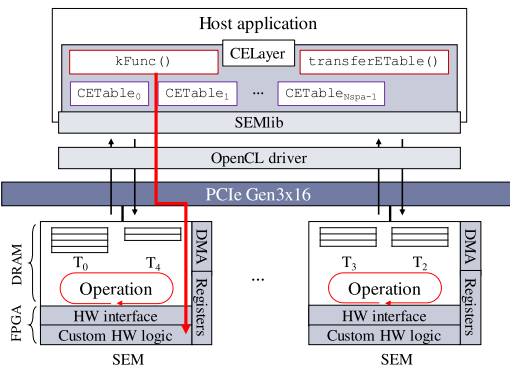


Fig. 2. The overall architecture of SEMS.

Authorized licensed use limited to: POSTECH Library. Downloaded on February 22, 2025 at 07:44:14 UTC from IEEE Xplore. Restrictions apply.

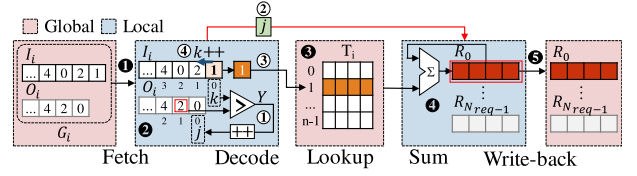


Fig. 3. The execution of RSes using SEMS.

parallelism (see Section 3.3). By invoking `CELayer's transferETable()` method, an embedding table is loaded to the internal DRAM of the designated SEM module. The offloaded operations are implemented in SEMS' FPGAs, and are executed by calling `CELayer's kFunc()` method. Regardless of how the embeddings are stored in the DRAM and how the offloaded operations are implemented, the device-level details are hidden behind SEMlib.

For fast prototyping of the offloaded operations, we use high-level synthesis (HLS) that converts C/C++ functions to register transfer level (RTL) code. SEMlib also works as a bridge between the software and the hardware; when a class function call arrives, it transfers the parameters to the hardware accelerator via OpenCL, supported by commercial FPGAs. It lets the direct communication between the host and SEM, bypassing the intermediate software and I/O layers.

When offloading the operations, we consider two aspects: (i) how much data transfer could be reduced and (ii) how easily the offloaded operations could be implemented. While keeping these in mind, we offload two operations, lookup and aggregation, for the embedding layer and feature interaction.

For each sparse feature, $N_{lookups}$ embeddings are aggregated into a vector (see Fig. 1 (right)). Thus, by offloading the lookup operation, an $N_{lookups} \times$ data reduction is achieved. SEMS further reduces the amount of data to transfer by offloading a part of feature interaction. As the associative law holds for addition, feature interaction could be split and performed in two levels, the SEM device and the host. Instead of sending the output vectors (i.e., N_{spa} vectors for N_{spa} sparse features) of the embedding layer to the host directly, we reduce them into a single vector and transfer it to the host by performing a part of feature interaction in the SEM side. This leads to an $N_{spa} \times$ data reduction. As a result, there is a total of $N_{lookups} \cdot N_{spa} \times$ data reduction (e.g., if $N_{lookups} = 120$ and $N_{spa} = 32$, $3,840 \times$ data reduction is accomplished). This can increase the effective bandwidth of PCIe by several thousand times. In the host side, the rest of feature interaction is applied to the outputs from the SEM devices and the bottom MLP.

The operations involved in the embedding layer and feature interaction are simple array indexing for embedding lookups and element-wise sum. They are simple to implement in FPGA, requiring a small amount of hardware resources.

3.2 Execution of Recommender Systems Using SEMS

We explain the detailed architecture of the hardware kernel (`kFunc()`) in Fig. 3. The kernel accepts two major input parameters for a mini-batch of requests, an array of indices I and an array of offsets O . The kernel processes one sparse feature at a time. For each sparse feature i , we form a group G_i , consisting of I_i and O_i , from the two arrays.

The two arrays transferred from the host are kept in the global memory shared by the host and SEM. For a better performance, the arrays in the global memory are fetched to the local memory (1 in Fig. 3). In the decode phase (2 in Fig. 3), to lookup an embedding to serve R_j , we decode G_i to obtain the request number j and the index of the embedding, using two counters: request counter and index counter. For R_j , embeddings whose indices are $I_i[k]$ for k in range $[O_i[j], O_i[j+1])$ are looked up. j starts from 0 and the request counter increases j by 1 when all the embeddings for R_j have been

looked up (① in Fig. 3). k starts from 0 and the index counter increases k by 1 at the end of each decode phase (④ in Fig. 3). Using $I_i[k]$ (③ in Fig. 3), the embedding at that index is retrieved from T_i during the lookup phase (⑤ in Fig. 3) and is stored locally.

Aggregation is operated in the sum phase (⑥ in Fig. 3). For each inference request, SEM produces an output vector. Thus, for a mini-batch of N_{req} requests, SEMS maintains N_{req} vectors. The elements of these vectors are initialized to zero. Once an embedding is retrieved, it is added in an element-wise manner to the vector for R_j . j obtained during the decode phase is used to identify this vector (② in Fig. 3). If a request requires multiple lookups, embeddings from subsequent lookups are added up to the same vector.

Once the three phases have been gone through for G_i , for all the requests in a mini-batch, the above steps are repeated for the next group, G_{i+1} . As the feature interaction operation is sum, the aggregated result for each sparse feature is again aggregated into a vector. Thus, for G_{i+1} , the retrieved embeddings are added in an element-wise manner to the same vector as did for G_i . In this way, aggregation and feature interaction are performed at once without any additional hardware logic.

3.3 Optimization

We optimize SEMS at three levels: (i) CU-level, (ii) SEM-level, and (iii) SEMS-level, where CU stands for compute unit, which is the unit of kernel execution.

CU-Level Optimization. Using HLS, we apply pipelining to the kernel. There are three major components in a single pipeline of the hardware kernel discussed in Section 3.2: decode, lookup, and sum. Since we designed the components to have no dependency between them, pipelining can be easily applied.

We apply two optimization techniques to further increase the impact of pipelining. First, we exploit the local memory to keep the length of each stage of the pipeline short. The local memory is only accessible from the kernel, but has a much shorter access time than the global memory. To exploit the fast access time of the local memory, we add a fetching phase (① of Fig. 3) which fetches G_i from the global memory to the local memory. During the decode phase, G_i is accessed from the local memory. For the sum phase, we manage the output vectors locally and add a write-back phase (⑥ of Fig. 3) that writes them back to the global memory. As the memory accesses during the fetching and the write-back phase are sequential, the data are transferred in bursts, showing a very small latency.

We also reduce the length of each pipeline stage by parallelizing the sum operations. In the sum phase, an element-wise sum is performed on two vectors. We parallelized the sum operations at different index positions as there is no dependency between them.

SEM-Level Optimization. SEMS creates multiple CUs per SEM and data parallelism is applied across them, equally distributing the requests in the mini-batch. Since there is no dependency between the requests, the CUs can run in parallel. Therefore, the throughput scales with the number of CUs. Moreover, as the CUs share the same memory space, unlike the typical data parallelism, the model does not need to be copied.

SEMS-Level Optimization. If a single SEM does not provide enough capacity to load all the embeddings, model parallelism could be applied, which exploits multiple SEMs. We propose a partitioning scheme called *table-wise partitioning*. It partitions the embedding layer by assigning each embedding table to SEM. As the feature interaction operation is sum, there is no dependency between the operations performed on each embedding table. Thus, each SEM can process the embedding layer and feature interaction on the assigned embedding tables, independent of another. Done with the dedicated computations, each SEM sends the (partial) output vector to the host. Then, these vectors are accumulated together in the host to produce the final output.

When partitioning, we need to consider both size and load balancing. Size balancing is balancing the sum of the sizes of the assigned tables between SEMs and load balancing is balancing the amount of computations between SEMs. A trade-off may exist between the two and in order to keep the inference time low, we give a higher priority to load balancing. As the number of lookups for each table is equal, the load of each SEM is determined by the number of tables assigned to SEM. For load balancing, we thus partition the embedding layer so that each SEM has the same number of tables, $\frac{N_{spa}}{N_{SEMS}}$, where N_{SEMS} is the number of SEM devices in SEMS. This not only balances the communication cost between each SEM and the host but also reduces it. For size balancing, we assign the embedding tables from largest to smallest one by one to the SEM that has the smallest sum of the sizes of the assigned tables.

4 EXPERIMENTS

4.1 Experimental Setup

We implement a proof-of-concept prototype of SEMS. The setup of SEMS is explained in Section 1. The host system contains a 64GB DDR4 memory, a 2TB SSD, and an Intel Xeon Gold 6248R CPU running at 3.00GHz with two NUMA nodes, 24 cores per CPU, and an RTX-3090 GPU with a 24GB GDDR6X memory.

We use DLRM [4] as the target recommendation system. The execution time of DLRM inference is used as the default evaluation metric which is calculated by adding up the elapsed time of each component for 1,000 iterations. The mini-batch size is 128. To understand how the number of embedding lookups per sparse feature affects the performance of SEMS, we use two different types of DLRM models: Model A and B. They share the same bottom MLP (13-512-256-128-128) and Top MLP (512-256-1) structure and $D = 128$. Model A uses Criteo's dataset [6] for the dense features and use synthetic dataset with 32 sparse features, where $N_{lookups} = 120$. Model B uses Criteo's dataset for both the dense and sparse features, having 26 sparse features, where $N_{lookups} = 1$. The size of Model A and B are 7.63GB and 16.1GB, respectively.

We compare SEMS with typical CPU-based and GPU-based systems where the entire model is loaded in the CPU memory and the GPU memory, respectively. Conversely, SEMS keeps the entire embedding layer in SEMs' DRAM and performs a part of feature interaction in SEMs. The top and bottom MLP and another part of feature interaction are run using the CPUs. We also evaluate the impact of optimization techniques.

For fast evaluation, we use models with a relatively small-sized embedding layer (7.63GB and 16.1GB). In reality, an embedding layer is expected to reach 1TB in size [7]. Considering that commodity servers have up to 128GB or 256GB of DRAM due to the limited number of DIMM slots, the Model-to-DRAM ratio is 8:1 or 4:1. To emulate this, we conduct another experiment using the CPU-based system where we limit the system memory size to 75%, 50%, 25%, and 12.5% of the embedding layer size. The CPU-based system cannot hold the entire embedding layer in the DRAM and relies on the OS swapping to extend the system memory, whereas in the GPU-based system, as swapping is not supported, the model cannot be run without multiple GPUs, which is very expensive. Thanks to the high scalability of PCIe, SEMS employs as many SEMs as possible to accommodate the embedding layer, in a cost-effective way.

4.2 Experimental Results

Figs. 4a and 4b present the experimental results of Model A and B, respectively. Full is a system with enough memory to fully load the entire embedding layer. $n\%$ represents the CPU-based system that has an insufficient amount of DRAM, holding only $n\%$ of the embedding layer. m CUs is SEMS with m CUs. The maximum

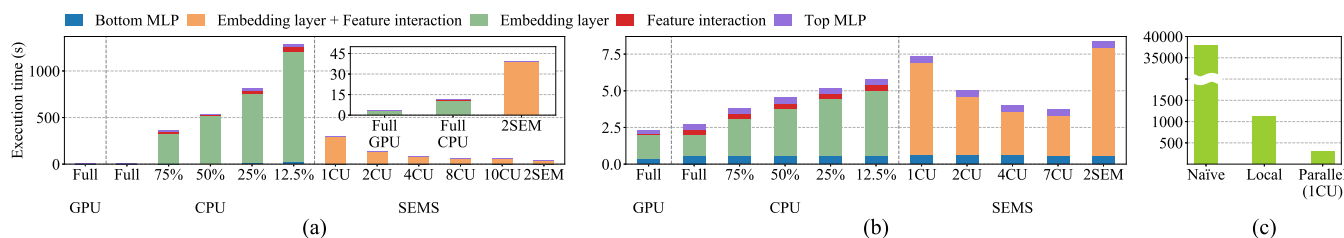


Fig. 4. The DLRM execution time. (a) A breakdown for Model A. (b) A breakdown for Model B. (c) The comparison between optimization techniques.

number of CUs that could be synthesized through HLS depends on the model. 10 and 7 CUs are created at maximum for Model A and B, respectively. 2SEM is SEMS with two SEMs where each SEM has the maximum number of CUs and table-wise partitioning is applied. In the experiments using SEMS, the CU-level optimization techniques are applied.

We compare SEMS under the most well-performing setting with the GPU and the CPU-based systems. In Fig. 4a 2SEM is about and 10.66 \times and 3.42 \times slower than Full1 (GPU) and Full1 (CPU), respectively. In Fig. 4b, 7CU is about 1.60 \times and 1.36 \times slower than Full1 (GPU) and Full1 (CPU), respectively. This is owing to the longer latency of the interconnect (DIMM versus PCIe) and the HLS-based accelerator designs (which are not highly optimized than the RTL-based ones). As the embedding layer gets bigger than the capacity of the DRAM and the costly swap in and outs occur, SEMS outperforms the CPU-based systems, exhibiting up to 32.85 \times and 1.56 \times better performance in Model A and B, respectively.

CU-Level Optimization. We evaluate the impact of pipelining using Model A, focusing on how much individual optimization techniques shorten the execution time. The results are shown in Fig. 4c. Naive is the execution time when no optimization technique is applied. Local and Parallel are about exploiting the local memory and parallelism, respectively. The techniques have been applied accumulatively. Local reduces the execution time by 33.59 \times by exploiting the fast local memory and the burst data transfer between the host and SEM. Parallel further reduces the execution time by 3.84 \times compared to Local.

SEM-Level Optimization. We analyze the performance of SEMS, varying the number of CUs. The performance improvement of SEMS becomes marginal as the number of CUs increases. This is because, due to the communication cost, there are some delays between the executions in different CUs. These delays prevent the executions from being overlapped fully. As the number of CUs increases, the execution time of each CU becomes shorter, decreasing the overlapped portion between the executions in different CUs.

SEMS-Level Optimization. We evaluate the scalability of SEMS with the number of SEMs. In Fig. 4a, 2SEM shows 1.48 \times faster performance compared to 10CU. In Fig. 4b, applying SEMS-level optimization to 7CU worsens the performance as the execution time of each CU is too short that executions in some CUs are not even overlapped and extra communication traffic occurs for the synchronization of two SEMs.

5 RELATED WORK

There have been several attempts to efficiently deal with large embedding layers. RecNMP [8] is a DIMM-based near-memory processing solution for DL-based RSes. By offloading the embedding layer to the DIMM and exploiting rank-level parallelism, RecNMP greatly improves the performance of DL-based RSes. However, since RecNMP is implemented in the buffer chip within a DIMM, a significant change in the HW is inevitable and customized CPU instructions must be added. SEMS presents a viable design option that could be deployed in conventional systems more easily by using commercially available DRAM+FPGA cards.

RecSSD [9] is an SSD-based near-data processing system for the inference of RSes. Similar to SEMS, it offloads the embedding layer and some of feature interaction to the SSD controllers. It also attempts to further reduce service latency by caching the embeddings in both the host and the SSD side. Although it outperforms conventional SSD-based systems and provides much higher capacity than SEMS, the long latency of NAND flash and the limited performance of ARM CPUs make RecSSD slower than DRAM-based systems like SEMS.

Heterogeneous architectures have been widely adopted to accelerate DL applications. [10] proposes a FPGA-GPU heterogeneous system for CNNs. It shows that direct hardware mapping (DHM) of CNN on an embedded FPGA can beat GPU-based processing but claims that FPGA cannot fully substitute GPU as DHM requires a lot of FPGA resources. [10] has a different scope from SEMS because it focuses on CNN-based DL applications rather than the embedding-based ones. However, [10] is in line with this study in that it shows the possibility of exploiting heterogeneity in accelerating DL applications.

6 CONCLUSION

We proposed SEMS, a scalable embedding memory system to accelerate DL applications with large embedding layers using FPGAs. The use of PCIe bus made SEMS highly scalable and versatile. The amount of data transferred was minimized by processing data in SEMs, improving the effective bandwidth of PCIe. We also optimized SEMS at various levels to maximize its benefits. The evaluation results show that SEMS was approximately 32.85 \times faster than a traditional CPU-based system. We plan to extend SEMS to support Intel's Compute Express Link (CXL), which lets the direct communication and memory sharing among CPU, SoC, GPU, and FPGAs.

REFERENCES

- [1] J. Dean, D. Patterson, and C. Young, "A new golden age in computer architecture: Empowering the machine-learning revolution," *IEEE Micro*, vol. 38, no. 2, pp. 21–29, Mar./Apr. 2018.
- [2] Xilinx, Inc., "Alveo U200 and U250 data center accelerator cards data sheet," 2021. [Online]. Available: <https://tinyurl.com/2p9xsw5f>
- [3] BittWare, "250-M2D M.2 accelerator module," 2022. [Online]. Available: <https://tinyurl.com/mwknws9x>
- [4] M. Naumov et al., "Deep learning recommendation model for personalization and recommendation systems," 2019, *arXiv:1906.00091*.
- [5] J. Park et al., "Deep learning inference in facebook data centers: Characterization, performance optimizations and hardware implications," 2018, *arXiv:1811.09886*.
- [6] Kaggle, "Criteo display advertising challenge dataset," 2014. [Online]. Available: <https://www.kaggle.com/c/criteo-display-ad-challenge>
- [7] E. K. Ardestani et al., "Supporting massive DLRM inference through software defined memory," *CoRR*, 2021, *arXiv:2110.11489*.
- [8] L. Ke et al., "RecNMP: Accelerating personalized recommendation with near-memory processing," in *Proc. IEEE/ACM 47th Annu. Int. Symp. Comput. Architect.*, 2020, pp. 790–803.
- [9] M. Wilkening et al., "RecSSD: Near data processing for solid state drive based recommendation inference," in *Proc. 26th ACM Int. Conf. Architect. Support Program. Lang. Operating Syst.*, 2021, pp. 717–729.
- [10] W. Carballo-Hernández et al., "Why is FPGA-GPU heterogeneity the best option for embedded deep neural networks?," 2021, *arXiv:2102.01343*.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.