Performance Model and Profile Guided Design of a High-Performance Session-Based Recommendation Engine

Ashwin Krishnan, Manoj Nambiar, Nupur Sumeet, Sana Iqbal
TCS Research, Mumbai, India
ashwin.krishnan1@tcs.com,m.nambiar@tcs.com,nupur.sumeet@tcs.com,sana.iqbal2@tcs.com

Abstract

Session-based recommendation (SBR) systems are widely used in transactional systems to make personalized recommendations to the end-user. In online retail systems, recommendations-based decisions need to be made at a very high rate especially during peak hours. The required computational workload is very high especially when there is a larger number of products involved. Session Based Recommendation (SBR) models incorporate the learning-based product buying pattern from various user interaction sessions and try to recommend the top-K products, the user is likely to purchase. These models comprise several functional layers that widely vary in their compute and data access patterns. To support high recommendation rates, all these layers need a performance optimal implementation, which can be a challenge given the diverse nature of the computations involved. For this reason, one compute platform - whether it is CPU, GPU, or a Field Programmable Gate Array (FPGA) may not be able to provide an optimal implementation for all the layers. In this paper, we describe performance modeling and profile-based design approach to arrive at an optimal implementation, comprising of the hybrid CPU, GPU, and FPGA platforms for NISER - a session-based recommendation model that avoids popularity bias in recommendations. In addition, the design for the CPU-FPGA hybrid platform is implemented for NISER and we observed that experimental results closely follow the results predicted by the performance model for the implemented deployment option.

CCS Concepts

• **Hardware → Testing with distributed and parallel systems**; • **General and reference → Performance**.

Keywords

Performance Modelling, Heterogeneous Architecture, Session-Based Recommendation Systems

ACM Reference Format:
Ashwin Krishnan, Manoj Nambiar, Nupur Sumeet, Sana Iqbal. 2022. Performance Model and Profile Guided Design of a High-Performance Session-Based Recommendation Engine. In *Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE '22), April 9–13, 2022, Bejing, China.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3489525.3511692

1 Introduction

Recommendation systems have become an important aspect on various online platforms like E-commerce and video serving enterprises to target the right products or videos to the right audience. It helps customers to have a better personalized experience in shopping or to suggest videos of interest. The enterprise in turn benefits by attracting more potential customers. Most of the traditional recommendations like collaborative filtering [7, 12, 13], Factorizing Personalized Markov Chains (FPMC) [24], and more recent Deep Recommendation-Based models [21, 36, 37] for Click-Through Rate (CTR) prediction use long-term users' interaction with the web and has user profiles. Here for a given user, thousands of potential items are passed through the recommender system to evaluate their probability of being clicked [10].

In many scenarios, user identities are anonymous and the length of the session is often short causing the conventional Recommendation Models (RMs) to under-perform [29, 35]. A Session-Based Recommendation (SBR) extracts users' preference that are hidden within individual sessions [29, 35]. Thus, SBR has attracted a lot of research for short-term recommendations. In E-commerce, the role of SBR is to recommend the top-K items (products) based on the sequence of items (products) clicked so far in the session [9]. Although, many models have been proposed earlier, the state-of-the-art SBR models [9, 22, 23, 29, 30, 32, 35] are Graph Neural Network (GNN [25]) based. This is because of GNN's ability to capture complex item transitions using graph-structured data that is fundamental for Recommender Systems [31]. Also, going forth, we are taking the example of E-commerce as the use case example for the SBR.

1.1 Challenges in Session-Based Recommendation System

A lot of study and work has been done for the inference acceleration for conventional Deep Recommendation-Based models [10, 11, 14, 15, 17, 19]. It has been found that random memory access while accessing the embeddings is the bottleneck in such models [17, 19]. Unlike conventional recommendation models, SBR models have a complex architecture with GNNs, attention layer, embedding layer, and final scoring layer as shown in Figure 1, each of which has unique compute and memory access behavior. As discussed in detail in section 4, we found that the GPU suffers from huge latency in performing array concatenation operation, transpose operation, and random memory access, whereas performs very well in dense computation. Also, GPU works better for larger batches of data than the real-time inference with batch size=1 [10]. The Field Programmable Gate Array (FPGA) with its heavy pipelining

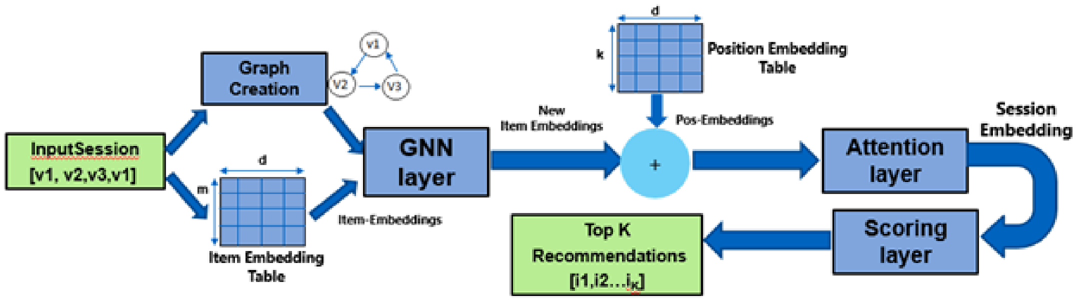


Figure 1: A Session-based Recommendation Model

ability has a great potential to accelerate the matrix-manipulation operations using its reconfigurable LUTs and accelerate random memory access using the on-chip Block RAMs (BRAMs) and High Bandwidth Memory (HBM).

The Conventional models for CTR prediction must meet the SLA or latency requirement which is in the order of tens of milliseconds [10]. So, the quality of recommendation depends on the total number of items explored within that set SLA that defines the batch size [14]. This is slightly different from the SBR models, where we obtain a session embedding [32] and compare it with all the items in the system to obtain top-K closest items. Here, in the SBR system, since the user arrival time would vary, batching is not very practical as we would need to wait for N users. Rather, a high-speed real-time inference architecture is more beneficial so that the current inference executes before the next use's arrival or minimizes the waiting period for the next inference.

In this paper, we have analyzed the real-time inference latency and bottleneck block for CPU, GPU, and FPGA. Our findings indicate that heterogeneous architecture can deliver the highest throughput for real-time recommendations. Our contributions in this paper include:

- To the best of our knowledge, this is the first paper that performs a detailed modelling and inference latency analysis for session based-recommendation model on CPU, GPU, and FPGA to obtain a pipelined heterogeneous architecture.
- We implemented CPU-FPGA hybrid architecture with a latency speed up of 6.1x compared to a baseline CPU for a real-time inference (batch size=1) and a performance (throughput) speed up of 5x for batched inference.
- The use of HBM to access item-embeddings in parallel by replicating it across multiple banks. Comparison with DDR memory implementation is also performed to show the advantage of using HBM over DDR for storing the item-embeddings of thousands of unique items.

2 RELATED WORK

Recently, a lot of importance has been given to hardware-based architecture to accelerate conventional RMs for CTR prediction especially to speed up the memory bottleneck. [8, 14, 15] are FPGA based architecture. Centaur [14] implements Intel's HARPv2 chiplet-based CPU-FPGA to accelerate Facebook's Deep Learning Recommendation Model (DLRM) [21] whereas NEUCHIPS [8] have used Intel Stratix FPGA to accelerate the DLRM. Microrec [15] has used the HBMs inside the FPGA to efficiently store many embedding tables. Their work accelerates Alibaba's RMs that have a higher number of embedding tables compared to DLRM. These works cannot be extended to SBR models as they do not incorporate complex operations like graph creation, GNN, attention, and scoring. [17–19] re-designs the Dynamic Random Access Memory (DRAM) at the micro-architecture level to accelerate the embedding gather and reduction operation within the DRAM. Tensordimm [19] introduces DIMM-level parallelism in DRAM in a disintegrated GPU. RecNMP [17] introduces a memory-side-caching on DRAM to exploit embedding entries with high locality efficiently. TRiM [18] further extends the previous concepts by introducing a fine-grained NDP architecture to increase the data transfer throughput in DRAM data paths. Again, these works cannot be extended to compute bound operations in SBR models since they are focused on speeding up the memory-bound embedding lookup and pooling operations alone.

A couple of works has presented inference schedulers for conventional RMs for CTR predictions. Deeprecsys [10] presents an inference scheduler for CPUs and GPUs to meet the SLA by analyzing the input query and arrival patterns. RecPipe [11] implements an inference scheduler that maps different stages of recommendations on the heterogeneous platform by analyzing the input query sizes. Also, they have designed a custom accelerator RPAccel for accelerating RMs for CTR predictions. They are not meant for SBR because they don't incorporate complex layers like GNNs, graph creation, and the scoring layer.

There is much literature available on boosting performance in terms of accuracy for SBR models but hardly any analysis was done to accelerate them. Almost all the recent SBR models [9, 22, 23, 29, 30, 32, 35] are built on top of SR-GNN [32] to achieve higher prediction accuracy. NISER [9] has incorporated position embedding after the GNN layer in [32] and L2 normalization of embeddings to overcome the popularity bias. SGNN-HN [22], FGNN [23] and DAGNN [35] alter the graph creation process. SGNN-HN uses Star Graph Neural Network to capture information between items that do not have a direct connection in the session whereas FGNN uses weighted-directed graphs to learn each item representation by aggregating its neighbors' embeddings with multi-head attention. DAGNN proposes demand-aware graph neural network to extract a session demand graph that learns the demand aware item embeddings. SRGI [30] includes global graph along with session graph to capture information from across the sessions. Also, here the position embedding is replaced with reversed position embedding.

3 BACKGROUND

A Session-Based Recommendation (SBR) model tries to predict the next item (or top K items) the user is most likely to click from an anonymous session with *n* items by modelling the complex transitions of items in the session. A Session-based Recommendation could be broadly divided into 6 different components (Figure 1). For details on the SBR model, the reader is encouraged to refer [9, 22, 23, 29, 30, 32, 35].

Item Embeddings [9, 22, 23, 29, 30, 32, 35]: The Recommendation Model (RM) consists of an item embedding table represented as a matrix M = $[i_1, i_2, \ldots, i_m]^T \in \mathbb{R}^{mxd}$ where an item embedding is a d-dimension vector for every unique item or product in the system. For a given set of *n* items (or products) in a session, corresponding item embeddings are fetched (it_emb $\in \mathbb{R}^{nxd}$) from the lookup table and fed as an input to the GNN layer.

Graph Creation [9, 22, 23, 29, 30, 32, 35]: This is the very first block in an SBR model where each session is converted into a graphical representation. For modelling purposes, we analyze the graph creation method proposed in [9, 32] where a graph is obtained by modelling sequential patterns over pair-wise adjacent items and then fed to the GNN block. The method is summarized in Algorithm 1.

Algorithm 1 Graph creation process

Inputs: $\{i_1, i_2, \ldots, i_n\}$: set of items in a session
Outputs: Normalized In-Out Adjacency matrices A1 and A2, respectively

```
1:  All the sessions are set to length 10 (k = 10) by padding zeros
    to the shorter sessions or by taking the last 10 items for the
    longer sessions.
2:  n =session_length (excluding the padded 0s)
3:  Obtain a list of unique items (unique_input) in the session.
4:  Create the array alias_input[n]. //Explained below
5:  Create the matrices UA_in [nxn] and UA_out [nxn] and initial-
    ize all elements to 0.
6:  for j in range (n) do
7:      u = alias_input[j] for j ≠ session_length-1
8:      v = alias_input [j+1] for j≠ 0
9:      UA_in (u, v) = 1
10: end for
11: UA_out = $UA\_in^T$
12: A1 = norm (UA_in) and A2 = norm (UA_out)
```

Here, in the Algorithm 1, the array alias_input is used to store the information about the original position of items which is lost after performing unique operation. For *e.g.*, Consider a session with sequence of items (or products) with ids {10,4,6,10,1}. On applying the get unique operation, we get a vector unique_input {1,4,6,10} which is a vector of unique items accessed in the session. Thus, alias_input stores the information {3,1,2,3,0} which means the 0^{th} position in the original input session is present at the 3^{rd} place in the unique_input vector. The alias_input array is used in reordering the output of GNN layer to get back the original sequence of items in the session.

The normalization (norm) operation on UA_in and UA_out is performed by summing the elements of an individual column and dividing the sum (if non-zero) by all the elements of that column. The normalized in-out adjacency matrices (A1, A2 $\in \mathbb{R}^{nxn}$) are generated to incorporate the graph structure [9, 31] and contains the information about incoming and outgoing edges for every node.

GNN [9, 22, 23, 29, 30, 32, 35]: This is one of the main components for an SBR, where we obtain a new representation for every item in the session referred as the new_item_embeddings (nie). It uses the session graph to capture complex item transitions. Gated graph neural network (GGNN), GAT and GCN are commonly used forms of GNNs [22]. GGNN [9, 22, 29, 30, 32] is the most used form of GNN which is governed by the following equations:

$$a1 = A1 * (it\_emb * H1 + b1) + b3 \tag{1}$$

$$a2 = A2 * (it\_emb * H2 + b2) + b4 \tag{2}$$

$$a = concatenate(a1, a2) \tag{3}$$

$$reset\_gate = \sigma((a * W1 + b5) + it\_emb * W2 + b6) \tag{4}$$

$$input\_gate = \sigma((a * W3 + b7) + it\_emb * W4 + b8) \tag{5}$$

$$new\_gate = tanh((a*W5+b9)+((it\_emb*W6+b10)*reset\_gate)) \tag{6}$$

$$nie = (new\_gate + input\_gate) \odot (it\_emb - new\_gate) \tag{7}$$

Here $W_a = [W1, W3, W5]^T \in \mathbb{R}^{2dx3d}$; $W_i = [W2, W4, W6]^T \in \mathbb{R}^{dx3d}$; b1-b10 $\in \mathbb{R}^{nxd}$ where the vector (1xd) is replicated n times to match dimensions for matrix addition; H1, H2 $\in \mathbb{R}^{dxd}$ are model parameters initialized upon training. $\odot$ denotes element-wise multiplication operator. Also, a $\in \mathbb{R}^{nx2d}$

Position Embedding Block [9, 29, 30]: The position embedding is a d-dimension vector representation for every existing position of the item in the session. This vector gets added to the output of GNN to incorporate the sequential information of item transactions in the session [9]. Algorithm 2 summarizes the position embedding block. In Algorithm 2, pos_emb $\in \mathbb{R}^{nxd}$. P[i][:] and pos_emb[i][:] denotes-accessing all the elements of the row 'i'.

Before adding the position embeddings (pos_emb) to the new_item_embeddings (nie), the re-ordering of the new_item_embeddings is performed to get the original sequence of inputs in the session.

The work presented in [29] and [30] has incorporated position vectors in reversed order (reversed position embedding) to achieve better accuracy.

Algorithm 2 Position Embedding Block

Inputs: nie and alias_input
Output: P: new item embeddings with incorporated sequential information of item transactions in the session

```
1: for i in range (n) do //n = session_length
2:     P[i] [:] = nie[alias_input[i]] [:] + pos_emb[i] [:]
3: end for
```

Attention Mechanism: This step provides us with a session embedding by considering both the global preference and the user's recent interest (latest item in the session). It is a d-dimension vector (sess_emb $\in \mathbb{R}^{1xd}$) representation of the item that the user is most likely to click next. The attention mechanism is governed by the following equations:

$$Q = \sigma(P_n * W1 + P * W2) \tag{8}$$

$$\alpha = \Sigma((Q * W_Q) \bigodot P) \tag{9}$$

$$sess\_emb = L2\_norm(concatenate(\alpha, P_n) * W + bias) \tag{10}$$

Here W1 and W2 $\in \mathbb{R}^{dxd}$, $W_Q \in \mathbb{R}^{dx1}$, W $\in \mathbb{R}^{2dxd}$, and bias $\in \mathbb{R}^{1xd}$ are model parameters. $P_n \in \mathbb{R}^{1xd}$ is the embedding of the last item in the session, and P $\in \mathbb{R}^{nxd}$ is the output of the position embedding block and Σ is the summation of rows of the matrix. $\alpha \in \mathbb{R}^{1xd}$ and Q $\in \mathbb{R}^{nxd}$. L2_norm is computed by summing the squares of individual elements of the vector and then dividing it with each element.

Scoring with top K recommendations: This is the final step where a score (cosine similarity) is calculated in the form of a dot product between all the item embeddings in the system with the newly obtained session embedding (Equation 10). At this point, every item has a score and the items with top K scores are recommended to the end-user.

3.1 NISER

NISER [9] is an SBR model that incorporates position embedding to effectively obtain position-aware item representations and L2 normalization of embeddings to overcome the popularity bias. The GNN layer is comprised of Gated Graph Neural Network (GGNN). The model achieves 53.39 Recall@20 and 18.72 Mean Reciprocal Rank (MRR@20) on Diginetica Dataset [1]. The value of d and k (in Algorithm 1) is set to 100 and 10, respectively.

4 MODEL EXECUTION ON CPU AND GPU

In this paper, we are considering NISER [9] as the baseline model and Diginetica [1] is taken as the baseline dataset for evaluating the model. The dataset has an average session length of 5.12 items and a total of m=43097 unique items . The value of the d is set to 100 which is consistent with other works [9, 23, 29, 30, 32, 35] and hence item_embeddings $\in \mathbb{R}^{100}$ and position_embedding $\in \mathbb{R}^{100}$.

Table 1: Performance analysis on various architecture for a real time inference

Metric	CPU	GPU	C-G
Graph Creation(ms)	0.454	3.582	0.454
GNN Computation(ms)	0.816	0.769	0.769
Position Embedding (ms)	0.190	0.247	0.247
Attention Layer(ms)	0.623	0.679	0.679
Scoring Block(ms)	0.277	0.195	0.195
Total Latency(ms)	2.36	5.47	2.34
Throughput(inferences/s)	423.73	182.7	431

We first performed the model analysis on a 56-core CPU with Hyper-Threading (HT). The model excluding the graph creation was implemented using the PyTorch [2] library. Python lists and NumPy were used to perform the graph creation operation. We found that for a real-time inference (Batch size=1), the overall latency is 2.36ms. From the first column in Table 1, we could observe that maximum time is consumed in GNN and the attention layer.

GPUs are well-known hardware accelerators. Model implementation on GPU is very simple owing to various GPU supported libraries like PyTorch. Just by introducing a few lines of code, one could map the model onto the GPU:

device = torch.device("cuda")
model.to(device)

We implemented the entire model on a V100 GPU using the PyTorch library that uses CUDA/cuDNN 10.1. To implement graph creation on GPU, all the NumPy arrays/matrices and Python lists were treated as PyTorch Tensors[2] and mapped onto the device 'Cuda'.

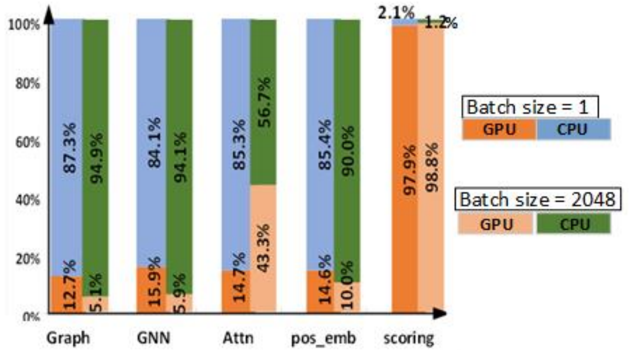


Figure 2: Profiling results on GPU for a batch size of 1 and batch size of 2048

From columns 1 and 2 of Table 1, we can observe that for a real-time inference with Batch size=1, the latency to implement GNN and scoring block on GPU is slightly lesser than that of CPU. For attention layer and position embedding layer, it is slightly higher than the CPU. The latency difference between CPU and GPU is not very significant for these layers for real-time inference. Whereas from Table 1 and Algorithm 1, we can observe that implementing graph creation process that consists of array and matrix operations like array creation, concatenation, matrix transpose, reshape, element-wise comparison, and a lot of non-streamlined memory access is faster on CPU (7.9x) than on GPU. This observation motivated us to implement the graph creation step on the CPU and the rest of the model on GPU to get the CPU-GPU (C-G) implementation to achieve better performance. Here, once the CPU implements the graph creation step, the CUDA memcpy HtoD is performed to transfer the data to the GPU. We could observe from Table 1 that for a real-time inference (Batch size=1), the latency of CPU- GPU (C-G) implementation is almost the same as that of CPU implementation.

Upon profiling with nvprof tool [6], we found that even for a pure GPU implementation, the actual time spent in GPU activities (GPU kernels) hardly exceeded 15% (Figure 2) of the total execution time for an individual block of operations. The rest of the time is consumed in other CPU activities within the PyTorch including CUDA API calls. Approximately, 87% time is consumed in CPU activities when the graph creation step is implemented using PyTorch Tensors that invoke a large number of CUDA API calls. Also, almost 23% of the total time is consumed in cudaMemcpy operations. Since PyTorch implementation of graph creation is not optimal, custom fine-tuning on GPU needs to be explored further. All the reported latency numbers are obtained by averaging over multiple runs and by ignoring the first few values to omit the startup latency.

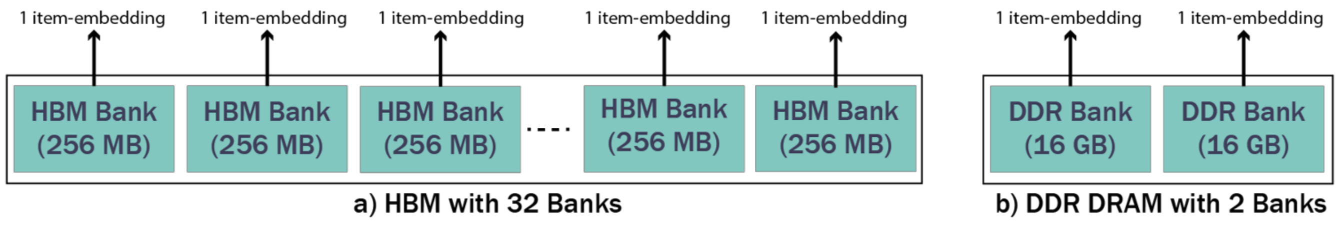


Figure 3: Memory interface for item-embeddings

5 DESIGN AND PERFORMANCE MODELLING ON FPGA

Acceleration of conventional recommendation models for CTR prediction using FPGAs [14, 15] show that there is a huge potential to accelerate SBR models using FPGAs. Xilinx Alveo U280 FPGA card [3] has large number of resources: approx. 1.3 million look-up Tables (LUTs) and 9000 DSPs capable of pipelined operations (Figure 4). The card also features an 8GB High Bandwidth Memory (HBM) system that can provide a bandwidth of up to 425 GB/s [28, 33]. Along with HBM, it has 16GB of DDR DRAM, and 8MB of on-chip BRAMs and 2607K registers that can be used to effectively store the item embeddings, position embeddings and model parameters. The on-chip memories could further be partitioned into smaller memory blocks that enable us to distribute the parameters across multiple partitions to access them in parallel. A careful prior analysis must be done to keep the trade-offs between resource utilization and maximum operational frequency that would impact the overall latency and throughput of the design. Before designing the architecture for individual blocks of SBR, we estimated the resource utilization and clock cycles required to perform a single floating point (FP32) operation at a particular frequency.

We performed modelling by taking 200MHz as the operating frequency and found that the FP32 multiplication and FP32 addition/subtraction consume two DSPs each. Also, each operation takes around 4 clock cycles and is pipelined with an initiation interval (II) of one clock cycle. The initiation interval of one (II=1) indicates that the input could be fed every clock cycle. The comparison operation takes 1 clock cycle with a resource utilization of 28 LUTs. These numbers may vary slightly for larger variations in frequencies. Thus, given a total of around 9000 DSPs available on the FPGA, at most 4500 FP32 multiplications and addition/subtraction operations (900 GFLOPS) could be performed concurrently. We also found that with an increase in the number of resources, maximum achievable frequency of operation reduces significantly due to routing constraints. For instance, at 90% of resource utilization, we could synthesize the system at only 80MHz. So, as a thumb rule, it is suggested not to go beyond 65-70% of overall resource utilization. Taking all these points into consideration we proceeded with modelling the layers of NISER on Alveo U280 [3].

5.1 Memory allocation for Item Embeddings

In a SBR system, n-item embeddings with each embedding $\in \mathbb{R}^d$ need to be fetched for a session with n items. So, we analyze whether parallelization can be leveraged for faster embedding access. The availability of 32 parallel HBM banks and 2 DDR DRAM banks on the Alveo U280 card allows us to replicate the embedding tables across 32+2=34 different banks and fetch them in parallel as shown in Figure 3a and 3b. [28] discusses that the setup read latency of DDR4 (worst-case latency of 32 clock cycles) is lower than that of HBM (worst-case latency of 62 clock cycles. Also, DDR4 memory

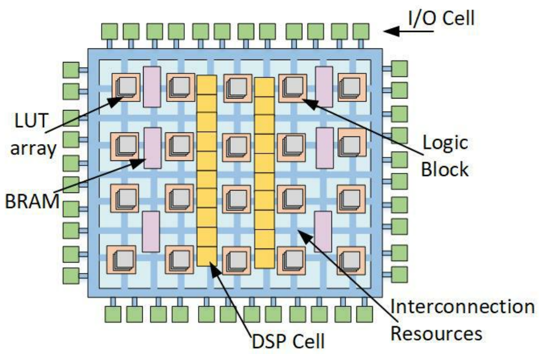


Figure 4: Internal Architecture of FPGA

controller could fetch 64 Bytes of data per bank in one clock cycle whereas one HBM bank could fetch only 32 Bytes per bank. Thus, for d = 100 (400 Bytes), a burst length of 7 and 13 would be required on DDR and HBM, respectively and the worst-case latency (page miss) to fetch an item embedding (400 Bytes) on DDR would be around 38 clock cycles whereas it would be around 74 clock cycles on HBM. Now, from Algorithm 1, we could observe that k (maximum possible session length) is set to ten. In other words, at most ten items embeddings need to be fetched for a single inference.

If we are to implement ten embedding fetch operations using the two DDR banks, it would require five roundtrips from each bank with a total worst-case latency of 190 clock cycles. Whereas it would take just one roundtrip to fetch ten item embeddings from ten different HBM banks, with an overall worst-case latency of 74 clock cycles. The HBM interface always fetches data in the multiple of 256 bits (32 Bytes) [33] but the size of an individual item embedding is 400 Bytes which is not a multiple of 32. So, we append zeros at the end to make it 416 Bytes. When reading the embeddings for inference, the last 16 Bytes are ignored. The embeddings are arranged in the exact order of items indices. In other words, for the 1st item, the corresponding embeddings occupy 1st 416 Bytes of space. Then for the 2nd item, its corresponding embeddings occupy next 416 Bytes of space, and so on. To fetch the embedding for the item with index 'i', we provide the address BA+(416*i) where BA is the base address of the HBM bank that is known in prior.

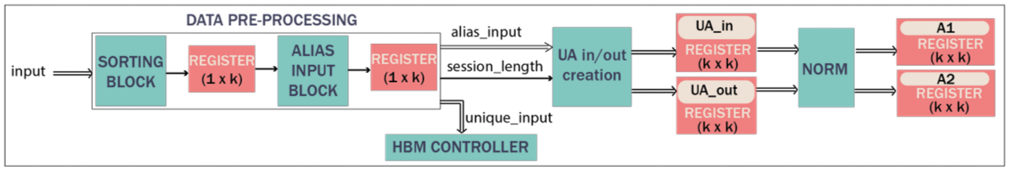


Figure 5: FPGA Architecture for Graph Creation

5.2 Graph Creation

Obtaining the unique items from the session as stated in Algorithm 1 is implemented by removing the repeated items in the linear time sorting network [27] and then storing the result in the registers. Calculating the session_length (n) is performed in parallel to the sorting operation. On FPGA, we cannot reshape the memory dynamically and since n (session length) in Algorithm 1 is a variable that would vary from session to session, therefore, the variable dimensionality n is replaced with the fixed dimensionality k from the Algorithm 1. This is being done for every vector and matrix in the system. Thus, the dimension of alias_input array is fixed to k and is created in k clock cycles. The 'for loop' in the Algorithm 1 is pipelined with II=1 and an iteration latency of three (one clock cycle for each line in the 'for loop') is achieved. Here, the loop is iterated k times instead of n and the registers for UA_in and UA_out are fixed to dimension (k x k) for the reason stated earlier. To avoid any extra information being captured in the graph from padded 0s for shorter sessions, we use a simple 'if' condition in the 'for loop' to specify that the UA_in and UA_out will be set only for iterations less than n and ignored for other iterations. This way the loop iterates till 'k', but information is captured in the graph for only n iterations. Also, both UA_in and UA_out are created in parallel as shown in Figure 5. For k=10, the total latency=10(alias_input creation) + 3(iteration latency) =13 clock cycles. Now, any pipelined path with 'inps' inputs, II=t1, and an iteration latency of t2, will have an overall latency described by equation 11.

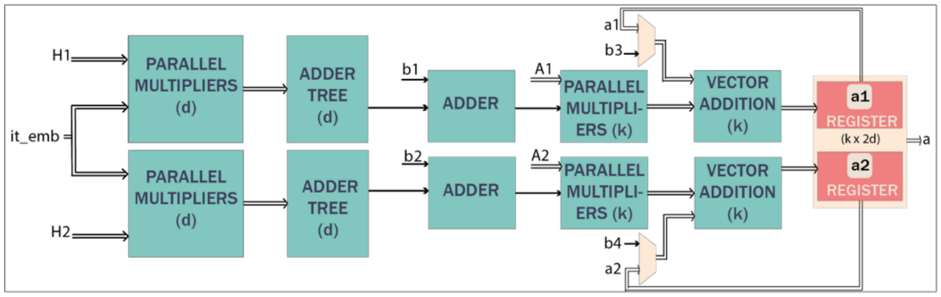
$$Latency = t2 + t1 * (inps - 1)\ clock\ cycles \tag{11}$$

The floating point (FP32) addition and division takes 4 and 5 clock cycles, respectively. So, to add ten numbers while performing normalization operation on UA_in and UA_out take 4*$log_2$10 = 4*4 clock cycles. Thus, normalization operation on an individual column pipelined with II=1 (t1=1) has an iteration latency (t2) of 4*4 + 5 = 21 clock cycles. Hence, to perform this operation on ten different columns (inps=10), the latency is 21 + (10-1) =30. Again, both the normalization operation is implemented in parallel.

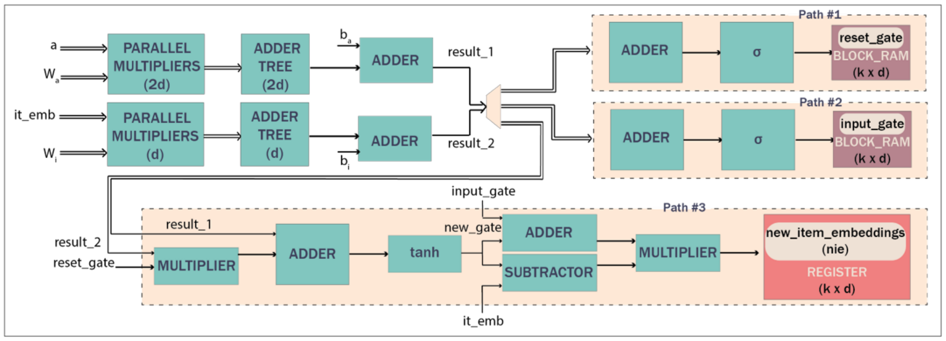
Unlike CPU/GPU implementation where item-embedding look-up operation is a part of GNN, here, the array 'unique_input' is sent to the HBM controller to fetch the item embeddings immediately after its creation. Here, the embedding look-up executes immediately after data pre-processing block (get_unique operation and alias_input creation) as shown in Figure 5. This happens in parallel to the 'for loop' and normalization operations. We add another 50 clock cycles while modelling as a safe margin to implement various control logic and load and store operations. Hence, the total estimated latency for the entire block in the Figure 5 to process is equal to pre-processing block latency + max(embedding lookup latency, (13+30)) + 50 = 20 + max(74, 43) + 50 = 144 clock cycles with a throughput of $1.4x10^6$ operations/s at 200 MHz.

5.3 GNN

All the model parameters are stored in the BRAMs to have low access latency. Any matrix multiplication ($A_{R1xC1} * B_{C1xC2}$) can be implemented using three loops. The outermost loop iterates R1 times, the middle loop C2 times and the innermost loop iterates C1 times. Throughout the system, for every matrix multiplication, the innermost loop is completely unrolled leaving only two loops (outer and center loops). At the hardware level, it can be visualized as C1-parallel multipliers followed by an adder tree unit. Thus, the total number of inputs (inps) is equal to the R1*C2 (For simplicity, inps-1 is be approximated with 'inps' in Equation 11). The iteration latency of adder tree unit is log_2R1 with II=1. The adder tree performs a summation of the elements of the vector. Figure 6a implements equations (1-3) with it_emb ∈ $\mathbb{R}^{kxd}$, H1/H2 ∈ $\mathbb{R}^{dxd}$ and A1/ A2 ∈ $\mathbb{R}^{kxk}$. Here dimensionality k is used instead of n for similar reason stated previously in graph creation sub-section. Thus, total number of inputs (inps) to the block is equal to d*k. In Figure 6a, the datapath for a1 and a2 is implemented in parallel. Now, the entire row of the it_em matrix gets multiplied with the corresponding elements in the column of the H1/H2 and sum of products (SOP) is achieved with an adder tree unit. To access all the elements of a column of H1/H2 in parallel, 'd' partitions of BRAMs are created and each row resides in an individual partition. A similar concept is applied to the rest of the blocks to access parameters in parallel. The output is then added with a bias ($b1$, $b2$). This takes around log_2d+2 (one for multiplication and one for bias-addition) clock cycles. Then a set of k-parallel multipliers and a vector addition unit incorporating 2 more clock cycles are used where an entire column of A1/A2 is multiplied with the same output, coming from the adder. For the very 1st iteration of the outer loop, the inputs to the vector addition unit is b3/b4. For the later iterations of the outer loop, i^{th} column ∈ $\mathbb{R}^d$ of 'a1' and 'a2' is fed back to the vector addition unit to achieve the SOP. Here, i is the i^{th} iteration of the center loop (keeping the convention of original matrix multiplication). Here, we could observe that with just two loops, we have implemented equations 1-3. The datapath is pipelined with an II=1 (t1=1). If we include the latency of 4 clock cycles for FP32 addition/multiplication at 200MHz, the overall iteration latency (t2) would be equal to 4*(log_2d+4) clock cycles. Thus, total latency of the block with d*k inputs is (d*k) + 4*(log_2d+4) clock cycles. After adding 50 clock cycles as a safe margin, for d=100 and k=10 (as mentioned in Algorithm 1) the overall estimated latency of the block in Figure 6a is 1094 clock cycles. We have ignored the term 'minus 1' in Equation 11 for simplicity.



(a) FPGA Architecture for GNN (Equations 1-3)



(b) FPGA Architecture for GNN (Equations 4-7)

Figure 6

In the Figure 6b, a ∈ $\mathbb{R}^{kx2d}$, W_a ∈ $\mathbb{R}^{2dx3d}$, it_emb ∈ $\mathbb{R}^{kxd}$ and W_i ∈ $\mathbb{R}^{dx3d}$. There are three paths (marked as path #1, path #2, and path #3 in Figure 6b) to implement equations 4-7 and all of them have been pipelined with II=1 (t1=1). Inputs to all these paths arrive from the shared 2d-parallel multipliers with an adder tree unit followed by an adder giving an overall latency of 4*($log_2$2d+1+1) clock cycles. Each of the k outer loop iterations will have a total of 3*d (for W_i) center loop iterations with d iteration for an individual path. The paths 1,2, and 3 are executed sequentially. Thus, total inputs(inps) to the block is 3*k*d. Now, the latency of sigmoid (σ) and tanh blocks are 12 clock cycles each (latency of FP32 exponential and division are 3 and 5 clock cycles, respectively at 200MHz). The iteration latency for path #1 and path #2 is 12+(4*1) = 16 clock cycles each. For path #3, the latency is 4*4 + 12 = 28 clock cycles. 4*4 is used for four adders/multipliers in sequence and 12 for the tanh block. Therefore, t2= (4*($log_2$2d+2) + 16 + 16 + 28) and the overall latency of the block is (3*d*k + 4$log_2$2d + 68) + 50 clock cycles. For d=100, the estimated latency is 3150 clock cycles. For the entire GNN operation, *i.e.*, Figure 6a and 6b together, the overall latency is (1094+3150) clock cycles, but in the pipelined implementation, the overall throughput is limited by the block with highest latency. Here, it is 3150 clock cycles (Figure 6a) and therefore the throughput at 200MHz is 63492 operations/s.

5.4 Position Embedding

The architecture for position embedding block is shown in Figure 7 where the pos_emb ∈ $\mathbb{R}^{kxd}$ is stored in the BRAM with d partitions and the 'for loop' in Algorithm 2 is pipelined with II=1. For every iteration in the loop, embedding vector ∈ $\mathbb{R}^d$ is fetched from the pos_emb. The lookup is performed by passing the position index to the pos_emb table similar to passing the item index to fetch item embeddings. Here, the 'i' in Algorithm 2 is treated as the position index. Since, alias_input ∈ $\mathbb{R}^k$, number of inputs to this block is k. Total iteration latency is equal to the latency of FP32 addition operation of 4 clock cycles. Thus, total estimated latency is (k+4)

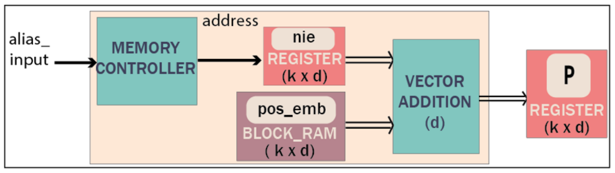
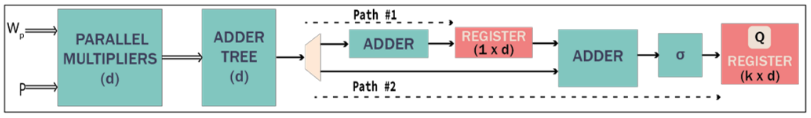


Figure 7: FPGA Architecture for Position embedding block

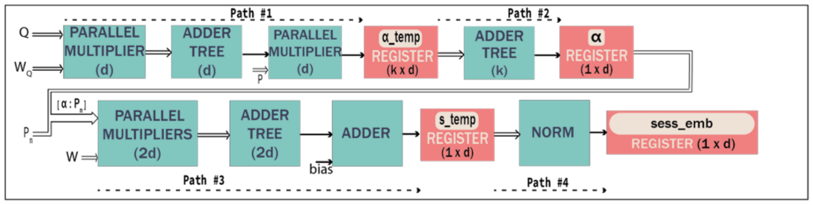
+10 which is equal to 24 clock cycles for k = 10. We added just 10 clock cycles as safe margin due to lower number of operations involved within the position embedding block.

5.5 Attention Layer

Figures 8a and 8b implements the equations 8-10. All the individual paths marked with dashed lines are pipelined with II=1. In Figure 8a, W_P = [W1,W2]^T ∈ $\mathbb{R}^{dx2d}$. With similar concept applied in GNN sub-section, we can observe that the latency of path #1 with d inputs and path #2 with k*d inputs in Figure 8a are d+4 and k*d+16 clock cycles, respectively. Hence, the overall estimated latency for Figure 8a is (k*d + d)+4*(log_2d +1) + (16+4) + 50 = 1202 clock cycles.



(a) FPGA Architecture for Attention layer (Equation 8)



(b) FPGA Architecture for Attention layer (Equations 9-10)

Figure 8

In Figure 8b, α_temp $\in \mathbb{R}^{kxd}$ and the division in norm takes 5 clock cycles. The overall estimated latency = latency of path #1 (inps=k) + latency of path #2 (inps=d) + latency of path #3 (inps=d) + latency of path #4 (inps=d) + safe margin latency=(k+4+4log_2d+4) + (d+4log_2k) + (d+4+ 4$log_2$2d+4) +(d+4+4log_2d+5)+50= 489 clock cycles. The Overall latency of attention layer is 1202+489 = 1691 clock cycles and throughput is 166.4x10^3 operations/s at 200MHz.

5.6 Scoring Block

To implement scoring block for the Diginetica [1] dataset with 43097 unique items involves the matrix multiplication of the sess_emb $\in \mathbb{R}^{1xd}$ with the entire set of item embeddings $\in \mathbb{R}^{dx43097}$. Here, total number of inputs is 43097. The iteration latency for d-parallel adders with adder tree is 4*(log_2d+1). Thus, the total estimated latency with d=100 is 43,129 clock cycles with a throughput of 4637 operations/s. This gives us 43097 distinct scores, out of which top-K scores need to be picked using a sorting or a top-K network. Taking reference from [20], we can observe that it consumes around 28 clock cycles to sort 128 elements using the sorting network. For K=20 [9, 22, 23, 29, 30, 32, 35], we need to feedback the top 20 scores from each of the 128 sorted scores.

Hence, effectively 108 scores could be passed through the sorting network giving a total of 43097/108=400 such iterations. For, the first iteration, dummy zeros could be treated as the feedback. Hence, to sort 43097 scores, it would take around 400*28 = 11200 clock cycles.

Hence, the overall all latency to implement this entire scoring block is 43,129+11200+50 (safe margin) = 54379 clock cycles. As stated earlier, the throughput of the entire scoring block is limited by the sub-block with maximum latency. Here, in this case, it is 4637 operations/s. Table 2 summarizes the modelling to estimate latency and throughput.

6 EVALUATION OF DEPLOYMENT OPTIONS

All the performance numbers in the CPU and GPU presented so far are for a single real-time inference *i.e.*, a batch size of one. We observed the CPU and GPU performance by varying the batch size, the results of which are displayed in Table 3a and 3b. The values in the table represent the throughput for each operation (batch_size/latency). The overall throughput with a layer-wise pipeline will be equal to that of the layer with minimum throughput. Some observations from the Table 3a and 3b are:

- There is an increase in overall throughput with an increase in batch size in both CPU and GPU implementations
- We can observe that the throughput of a single inference (batch size=1) impacts the GPU performance more, as compared to CPU
- Apart from graph creation, all the layers show very high speed-up compared to the CPU when the batch size is increased from 1 to 512
- Graph creation and position embedding layers run faster on CPU compared to the GPU. Rest of the layers run faster on the GPU.
- The graph creation is the slowest running layer for both CPU and GPU platforms

It follows that, on a server with GPU, it will be better to deploy the Graph creation layer on the CPU while the remaining layers are best deployed on the GPU. It may be desirable, to also deploy the position embedding layer on the CPU, given the higher throughput. However, the graph creation deployed in the CPU is still the bottleneck layer that would limit the overall throughput. Hence, it does not make much sense to take the development effort in deploying position embedding back on the CPU instead of the GPU. This configuration (C-G) *i.e.*, graph creation on the CPU and other layers on the GPU was experimentally evaluated and it yielded an overall throughput of 6795 inferences per second for a batch size of 2048 (Table 1-column 3 presents experimental evaluation for B=1). This implementation is not layer-wise pipelined and as a result, the overall throughput is less than 7635 (pipelined throughput in Table

Table 2: Estimated performance for individual layers

Layer	Estimated Latency (clock cycles)	Estimated Throughput (@200MHz)
Graph Creation	144	1.4x10^6
GNN Fig 5a	1094	63492
GNN Fig 5b	3150	
Position Embedding	24	8.3x10^6
Attention Fig 7a	1202	166.4x10^3
Attention Fig 7b	489	
Scoring	54379	4637

3a) inferences per second. This is quite close to what we can predict using Equation 12 on the data available in Table 3 and Table 4. For a given batch size (B) and throughput (x_i) for layer i, we can compute the non-pipelined throughput X_{np} with total number layers n_l as:

$$X_{np} = [\sum_{i=1}^{n_l} [x_i]^{-1}]^{-1} \tag{12}$$

With the implementation of a 2-stage pipeline, with one process implementing the graph creation layer on CPU and another process running the GPU implementation of the remaining layers, it should be possible to see a throughput that is equal to the throughput of the graph creation layer *i.e.*, 7635 inferences per second, assuming there is no PCIe communication bottleneck between GPU and CPU. When we compare the execution profiles in Figure 2 (batch size=2048), we can see that the GPU contribution to computational latency in the attention layer rises to 40% with the larger batch size. In the scoring layer, it stays close to 100%. Moreover, the GPU implementation of the scoring layer seems to deliver the highest throughput compared to what was measured on CPU and estimated with the modelling on FPGA. This makes the GPU be the preferred platform to run the scoring layer.

One interesting observation from the GPU profile is that the GPU contribution to the computational latency of the GNN layer is lesser for larger batches. This in turn means that CPU component of the computation is growing in the PyTorch implementation of GNN with an increase in batch size. Also, we can observe that there is an increase in the overall throughput of the GNN layer with batch size. This can be attributed to the increase in computational efficiency of the CPU component in the GNN, which is also gaining performance from the larger batch size.

Based on profiling and modelling, we can predict the best way to deploy various layers of NISER on different hardware blocks for maximum throughput as shown in Table 4. The table presents the deployment options in columns ranked in descending order of achievable throughput. For FPGAs, we do not assume any increase in throughput with a larger batch size.

Assuming that there is only one instance of each layer and all the layers are connected in a pipeline, the maximum achievable throughput (Max Thr) is reported to be that of the slowest. The lower bound on latency (LLB) is calculated as the sum of the inverse of the measured (CPU, GPU from Table 3a and 3b) and modeled (FPGA from Table 1) throughputs for the batch size of 2048. The last row of the table is the maximum interconnection network bandwidth required (NBR) to support the expected throughput. This is calculated by analyzing the size of the required data to be passed across the successive layers not deployed on the same hardware and the end-to-end pipelined throughput to be supported.

In the best ranked option, we could have each of the layers deployed to the hardware platform in which it is expected to perform the best as seen from the profiling and modeling results. However, in this configuration, we see many hops in the pipeline between the FPGA and the GPU. This could be a potential problem with the network interconnecting the FPGA and the GPU. There are two scenarios here:

(1) FPGA and GPU present in the same server connected over PCIe v3 network (15 GB/s bandwidth in one direction)
(2) FPGA and GPU installed in different host servers connect over 10 Gbps Ethernet network.

Table 3: CPU and GPU performance in terms of operations/s on various batch sizes (B)

(a)CPU

Layer	B=1	B=512	B=1024	B=2048
Graph Creation	2202	7448	7137	7635
GNN	1225	22790	45580	64194
Position Embedding	5263	23813	47627	57496
Attention Layer	1605	62271	124500	294500
Scoring Block	4405	16623	33247	33240
Pipelined throughput	**1225**	**7448**	**7134**	**7635**

(b)GPU

Layer	B=1	B=512	B=1024	B=2048
Graph Creation	279	402	401	404
GNN	1300.3	$533.8x10^3$	$1.1x10^6$	$2.1x10^6$
Position Embedding	4048.5	38380.8	41373.7	41558
Attention Layer	1472.8	$682.6x10^3$	$1.4x10^6$	$2.7x10^6$
Scoring Block	5434.8	$2.5x10^6$	$5.6x10^6$	$10.7x10^6$
Pipelined throughput	**279**	**402**	**401**	**404**

Table 4: Deployment considerations (B=2048)

Layer	Option 1 (Best)	Option 2	Option 3	Option 4
Graph Creation	FPGA	FPGA	FPGA	CPU
GNN	GPU	FPGA	FPGA	GPU
Position Embedding	FPGA	FPGA	FPGA	GPU
Attention Layer	GPU	GPU	FPGA	GPU
Scoring and Sorting	GPU	GPU	CPU	GPU
Max Thr (ops/s)	$1.4x10^6$	**63492**	**33240**	**7635**
LLB (ms)	**3.6**	**34.9**	**107.9**	**319.4**
NBR (MB/s)	**8.19**	**8.19**	**0.819**	**1.64**

Max Thr - Maximum Achievable Throughput, LLB - Lower Bound on latency, NBR - Network Bandwidth Requirement

In both the above cases, we have analyzed the data transfer requirements and it has been determined that the maximum network bandwidth requirement is much less than 10 Gbps. As a result, in both cases (PCIe or Ethernet), the network bandwidth will not limit the overall predicted inferencing rate. However, in deployment option 2, latency can be of concern and need to be evaluated against permissible latency per inference limits (SLA[10]). Other issues with the best modelled option are:

(1) The actual transfer speed achievable will be a function of the batch size for the incoming inference workload. One cannot assume that the entire PCIe or Ethernet bandwidth will be available for small batches.
(2) The GPU cannot concurrently process more than one layer at a time. So, multiple GPUs will be required to achieve the overall best-case throughput, assuming full pipelining.
(3) FPGA, like the GPU, has only 1 PCIe interface. Hence, some form of multiplexing will be required to serve Graph creation and position embedding workloads on the same FPGA. The FPGA cards may have more than one Ethernet link that could be used but do not make the design any simpler. Although using the High-Level Synthesis (HLS[34]), the designing becomes much simpler, but, we observed that sometimes, it is not as optimal as the custom RTL code.

(4) Profiling of GPUs (Figure 2) showed that for smaller batch sizes maximum time is spent in API and other CPU activities. The actual GPU utilization is very low. Hence, for smaller workloads, GPUs are not very efficient.

Often, especially with deep learning models, there is a tradeoff between latency and throughput [10]. From Table 3a and 3b, we observed that with a batch size of 1, we get the best latency but poor throughput and vice versa. One way to arrive at an optimal tradeoff will be to clamp down the maximum latency (SLA[10]) to a fixed amount of time, say 100ms (would depend on the system). For an end-user, the response time of say 2 seconds, 100ms could be a good upper limit for computing inferences and keep a good buffer for queuing delays and other overheads in the system. If we assume the FPGAs and GPUs are located in the same server, the first and second options hold very well as the PCIe transfer latencies would be comparatively negligible. Even the third option comes quite close to this limit. Other points to note regarding the deployment options are as follows:

- Even if the GPU cannot concurrently (without virtualization) compute more than one layer, not more than one GPU needs to be deployed in any of the deployment options listed in Table 4. This is because the service demand (across all batch sizes) for the GPU in those options does not exceed that of other resources (FPGAs and CPUs). Nevertheless, there may be other implementation overheads that could make this point invalid.
- The actual transfer speed achievable will be a function of the batch size for the incoming inference workload. One cannot assume that the entire PCIe or Ethernet bandwidth will be available for smaller batches.

The authors did not have exclusive access to a server with both FPGA and GPU for conducting further experiments at the time of this writing. For this reason, we select the third best option and present the performance evaluation in the next section.

7 EXPERIMENTAL EVALUATION ON FPGA

Alveo U280 card mounted on a 56 core CPU with HT and 256 GB memory is used to perform the experiments. We followed the Vitis accelerated flow [16] that provides all the necessary drivers (Xilinx XRT) to establish communication with the FPGA card. The host side code is written in C++ with the support of OpenCL [26] libraries to communicate with the hardware kernels present on an FPGA device. The data pre-processing block shown in Figure 5 is implemented using Verilog HDL in Xilinx Vivado and the rest of the pre-scoring blocks are implemented using Xilinx Vitis High-Level Synthesis [34] version 2019.2. The HLS code is converted into an IP and invoked in the Vivado to create the hardware kernel [34]. The maximum achievable operating frequency observed was 173.5MHz instead of 200MHz. This happens due to routing constraints. Table 5 summarizes the synthesis report from the HLS compiler. We could observe that the latency in terms of clock cycles matches closely with our modelling. Slight variation in the number of clock cycles in Table 2 and 5 is because of the change in actual frequency of operation and a slight mismatch with the assumed safe margin. Extra clock cycles in the attention layer (Figure 8b) is because the HLS compiler could implement the normalization block with an initiation interval of four instead of one. This adds around 400 (d*4) clock cycles [1] for d=100. The pre-scoring block consumes an overall of 42% LUTs, 41% DSPs, 24% registers, and 37% of total BRAMs available on Alveo U280.

Table 5: Achieved performance for layers on FPGA

Layer	Latency (clock cycles)	Latency (in μs)	Achieved Throughput
Graph Creation	176	1.01	$990x10^3$
GNN Fig 5a	1109	6.39	55401
GNN Fig 5b	3133	18.05	
Position Embedding	15	0.086	$11.6x10^6$
Attention Layer Fig 7a	1185	6.83	146413
Attention Layer Fig 7b	887	5.11	
Overall	**6505**	**37.5**	**26666**

7.1 Profiling embedding lookup performance on HBM

As stated in the modelling section (section 5.1), entire set of item-embeddings for the Diginetica dataset (17.9MB after padding zeros) is replicated across 10 HBM banks as shown in Figure 3a. Each HBM bank is connected to a Memory-Mapped AXI interface. The item embeddings are transferred to the HBM memory directly by calling the OpenCL function clEnqueueMigrateMemObjects [26] for each of the ten memory banks without running the kernel. The total number of clock cycles spent to perform the embedding lookup operation is calculated by reading the value of the counter placed in the FPGA kernel. The counter is enabled once the read embedding signal is triggered and stopped after receiving the last byte. We found that on average it takes 69 clock cycles to perform ten embedding fetch operations in parallel close to what we predicted from modelling.

7.2 An end-to-end inference on FPGA-CPU (F-C) hybrid

As shown in Figure 9, the item indices are the inputs that are transferred from CPU to FPGA via PCIe to give back the session embedding that is the output of the pre-scoring block. As discussed in section 6 (option 3) in Table 4, the scoring block is implemented on the CPU. We have used Intel MKL [5] library to implement the matrix multiplication and GNU's GSL [4] library to perform the top-K operation on the CPU. The Vitis hardware development platform does not yet support direct streaming from the host server to the Xilinx U280 FPGA due to which the kernel needs to be launched (instead of simply transferring the inputs to an always running kernel) for every session arrival adding a penalty of almost 120µs. Here, for every real-time inference, the kernel is launched using the OpenCL function clEnqueueTask. The items indices for a session are sent as unidirectional scalar inputs [16] to the device. Also, to retrieve the session embedding back from the device, the embeddings are first stored in the global memory (HBM) and then transferred to the host using clEnqueueMigrateMemObjects [26] function. Both these OpenCL functions together have an average overhead latency of

[1]The clock cycles overhead introduced due to initiation interval of 4 could be removed by enabling the unsafe math optimization option in the HLS compiler. However, the authors have not used this option at the time of this writing.

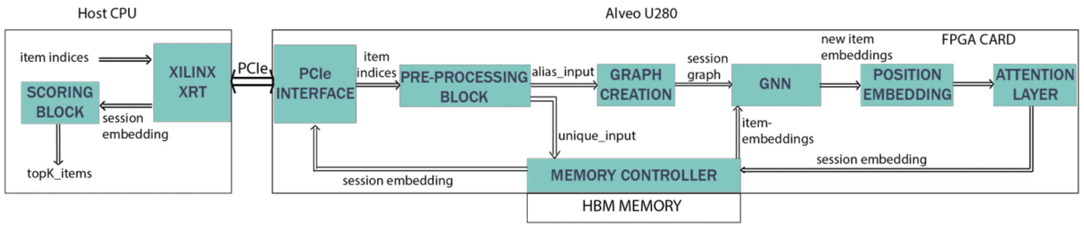


Figure 9: System Architecture

120µs limiting the throughput of FPGA to 8333 operations/s. The overall F-C implementation for a real-time inference of batch size=1 has a latency of 385μs (throughput=2597inferences/s) with a speed up (latency) of 6.1x compared to baseline CPU, 14.2x compared to baseline GPU and 6.1x compared to C-G implementation (Table 1) without any loss in accuracy. This is summarized in Figure 10.

To implement the batch inference, we transfer an entire batch (B) of sessions at one go to the HBM bank using clEnqueueMigrateMemObjects command. At the FPGA end, one session would be retrieved, processed and then the next session would be retrieved. The outputs of all these sessions are stored in an HBM bank. Then at one go, all the session embeddings are transferred back to the CPU. The overall throughput observed to compute the session embeddings of a batch of 2048 sessions on FPGA is 26666 inferences/s (Table 5) which is equal to the throughput of the entire system. By batching the inputs, we are able to hide the overhead latency of 120μs in calling the OpenCL functions and improve the throughput from 8333 to 26666 operations/s.

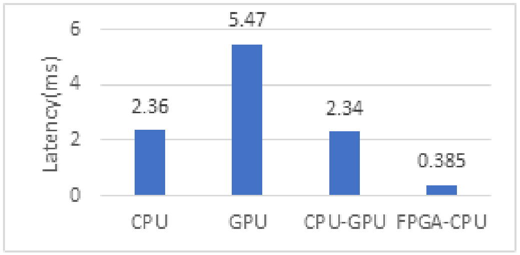


Figure 10: Latency comparison for a single inference on various architecture

8 DISCUSSIONS

From our experiments and modelling, we found that for a real-time inference in the SBR model, a heterogeneous architecture is the most efficient. The F-C architecture gave a speedup of 6.1x, 14.2x, and 6.1x compared to CPU, GPU, and C-G implementations, respectively for single inference with batch size=1. As stated in the previous section, the pre-processing block is implemented in Xilinx Vivado and rest pre-scoring blocks using Xilinx High-Level Synthesis (HLS) instead of creating the entire pre-scoring block on HLS. This was done because the HLS compiler 2019.2 was not able to realize any adder tree when we tried to create the kernel directly from the HLS instead of Vivado (the same code when treated as an IP was able to realize the adder tree). Because of this, we were able to send the next session input only after the complete processing of the previous session and we observed the FPGA throughput of 26666 operations/s instead of pipelined throughput of 55401 operations/s (Table 5) for pre-scoring blocks. Hence, for a batched (B=2048) F-C (option 3 in Table 4) implementation, the achieved throughput is 26666 inferences/s instead of estimated 33240 inferences/s. This implementation has a speed up (throughput) of 3.9x compared to C-G (6795 inferences/s) and 5x compared to baseline CPU (5055 inferences/s) implementation.

Although F-C implementation offers an inference speedup over baseline CPU and C-G, the throughput can be further improved by using the techniques mentioned below:

(1) By splitting the HLS IP into multiple smaller IPs.
(2) Using more than a single set of parallel multipliers with an adder tree for individual matrix multiplication. This would depend on the dimensionality of the d and the maximum session length (k) set for the system.

The aforementioned techniques would help in closing the gap between the achieved and estimated throughput. The authors intend to explore these options in the future. Our modelling approach could be extended to other SBR models by modelling other graph creation techniques used in [22, 23, 35] and any additional layer like GAT in [23].

9 CONCLUSION

In this paper, we have demonstrated that a mix of profiling and modeling methods can contribute towards building high-performance systems. Profiling is easy to perform on software-based platforms like CPU and GPU which have matured development frameworks and toolsets. On systems like FPGA, implementation risks are higher and hence, it is important to model the performance prior to implementation. Data from profiling and performance modeling were combined to analyze and make implementation decisions. In an industrial setup, development costs must also be thrown into the mix towards meeting high-performance objectives.

It is important to note that-what was modelled and later implemented on the FPGA was a custom solution for the given model being implemented. Whereas on the CPU and GPU, we used a matured framework, (PyTorch in this instance), which made it easy to implement the model. However, the PyTorch implementation was far from a custom implementation. Given the high percentage of time spent in the CPU in the PyTorch GPU implementation, we get the feeling that a custom implementation would have helped in the GPU base deployment as well. Nevertheless, in this case, the development would not be as straightforward as it has been on PyTorch. In the FPGA, we had the benefit of using the HLS framework which enabled implementing the entire model in C, which allowed a faster development and implementation. However, with HLS, we suspected that we may not get the desired performance as an HDL implementation would, and for the initial HLS implementation this was indeed the case. Thanks to the performance model that was developed earlier this could be verified and we were able to find a simple workaround that enabled us to achieve the desired predicted performance. The learning here was that synthesizing smaller functions independently and then integrating them together was an easier method to get the predicted performance as opposed to a single compilation of a larger code at once.

For the recommendation model being implemented for a real time inference with batch size=1, we observed that deploying the layers on CPU and FPGA, helps realize a 6.1x reduction in the latency as compared to the CPU-GPU deployment combination which was initially evaluated. With batching, the current F-C implementation has almost 4x speed up (throughput) compared to the CPU-GPU implementation (graph creation on CPU, rest on GPU) for even the larger batches of 2048. We notice that FPGAs with HBM memories are very helpful in achieving low latencies and high throughput thanks to several parallel ports and data replication. This does not necessarily mean that GPUs would not be able to achieve the same performance. As GPUs have similar HBM technology and with many streaming processors, it should be possible to achieve comparable performance on the GPU as well for retrieving embeddings. This will be addressed in future work along with other modelled deployment options that will include the evaluation of the present-day state-of-the-art approaches for the same.

References

[1] 2016. *CIKM Cup 2016 Track 2: Personalized E-Commerce Search Challenge.* https://competitions.codalab.org/competitions/11161

[2] 2019. *PYTORCH DOCUMENTATION.* https://pytorch.org/docs/stable/index.html

[3] 2021. *Alveo u280 data center accelerator card.* https://www.xilinx.com/content/dam/xilinx/support/documentation/data_sheets/ds963-u280.pdf

[4] 2021. *GSL - GNU Scientific Library.* https://www.gnu.org/software/gsl/

[5] 2021. *Intel-Optimized Math Library for Numerical Computing.* https://www.intel.com/content/www/us/en/develop/documentation/get-started-with-mkl-for-dpcpp/top.html

[6] 2022. *CUDA Toolkit Documentation v11.4.0.* https://docs.nvidia.com/cuda/pdf/CUDA_Profiler_Users_Guide.pdf

[7] Wanyu Chen, Fei Cai, Honghui Chen, and Maarten De Rijke. 2019. Joint neural collaborative filtering for recommender systems. *ACM Transactions on Information Systems (TOIS)* 37, 4 (2019), 1–30.

[8] N. Corp. 2020. *Neuchips recommendation accelerator recaccel.* https://2ca8d951-4386-4e41-9cab-50c86da5f5a8.filesusr.com/ugd/d79931_9382d53600f54d21a6eabe46d1f0ffa2.pdf

[9] Priyanka Gupta, Diksha Garg, Pankaj Malhotra, Lovekesh Vig, and Gautam Shroff. 2019. NISER: Normalized item and session representations to handle popularity bias. *arXiv preprint arXiv:1909.04276* (2019).

[10] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. 2020. 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)*. IEEE, 982–995.

[11] Udit Gupta, Samuel Hsia, Mark Wilkening, Javin Pombra, Hsien-Hsin S Lee, Gu-Yeon Wei, Carole-Jean Wu, David Brooks, et al. 2021. RecPipe: Co-designing Models and Hardware to Jointly Optimize Recommendation Quality and Performance. *arXiv preprint arXiv:2105.08820* (2021).

[12] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *Proceedings of the 26th international conference on world wide web*. 173–182.

[13] Jonathan L Herlocker, Joseph A Konstan, Loren G Terveen, and John T Riedl. 2004. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems (TOIS)* 22, 1 (2004), 5–53.

[14] Ranggi Hwang, Taehun Kim, Youngeun Kwon, and Minsoo Rhu. 2020. Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 968–981.

[15] Wenqi Jiang, Zhenhao He, Shuai Zhang, Thomas B Preußer, Kai Zeng, Liang Feng, Jiansong Zhang, Tongxuan Liu, Yong Li, Jingren Zhou, et al. 2021. MicroRec: efficient recommendation inference by hardware and data structure solutions. *Proceedings of Machine Learning and Systems* 3 (2021).

[16] Vinod Kathail. 2020. Xilinx Vitis Unified Software Platform. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Association for Computing Machinery, New York, NY, USA, 173–174. https://doi.org/10.1145/3373087.3375887

[17] Liu Ke, Udit Gupta, Benjamin Youngjae Cho, David Brooks, Vikas Chandra, Utku Diril, Amin Firoozshahian, Kim Hazelwood, Bill Jia, Hsien-Hsin S Lee, et al. 2020. Recnmp: Accelerating personalized recommendation with near-memory processing. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 790–803.

[18] Byeongho Kim, Jaehyun Park, Eojin Lee, Minsoo Rhu, and Jung Ho Ahn. 2020. TRiM: Tensor Reduction in Memory. *IEEE Computer Architecture Letters* PP (12 2020), 1–1. https://doi.org/10.1109/LCA.2020.3042805

[19] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. 2019. 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*. 740–753.

[20] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2012. Sorting networks on FPGAs. *VLDB J.* 21 (02 2012), 1–23. https://doi.org/10.1007/s00778-011-0232-z

[21] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G Azzolini, et al. 2019. Deep learning recommendation model for personalization and recommendation systems. *arXiv preprint arXiv:1906.00091* (2019).

[22] Zhiqiang Pan, Fei Cai, Wanyu Chen, Honghui Chen, and Maarten de Rijke. 2020. Star graph neural networks for session-based recommendation. , 1195–1204 pages.

[23] Ruihong Qiu, Jingjing Li, Zi Huang, and Hongzhi Yin. 2019. Rethinking the item order in session-based recommendation with graph neural networks. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*. 579–588.

[24] Steffen Rendle, Christoph Freudenthaler, and Lars Schmidt-Thieme. 2010. Factorizing personalized markov chains for next-basket recommendation. In *Proceedings of the 19th international conference on World wide web*. 811–820.

[25] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80.

[26] John E. Stone, David Gohara, and Guochun Shi. 2010. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. *Computing in Science Engineering* 12, 3 (2010), 66–73. https://doi.org/10.1109/MCSE.2010.69

[27] Joshua Vasquez. 2016. *SORT FASTER WITH FPGAS.* https://hackaday.com/2016/01/20/a-linear-time-sorting-algorithm-for-fpgas/

[28] Zeke Wang, Hongjing Huang, Jie Zhang, and Gustavo Alonso. 2020. Benchmarking High Bandwidth Memory on FPGAs. *arXiv preprint arXiv:2005.04324* (2020).

[29] Ziyang Wang, Wei Wei, Gao Cong, Xiao-Li Li, Xian-Ling Mao, and Minghui Qiu. 2020. Global context enhanced graph neural networks for session-based recommendation. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 169–178.

[30] Ziyang Wang, Wei Wei, Gao Cong, Xiao-Li Li, Xian-Ling Mao, Minghui Qiu, and Shanshan Feng. 2020. Exploring Global Information for Session-based Recommendation. *arXiv preprint arXiv:2011.10173* (2020).

[31] Shiwen Wu, Fei Sun, Wentao Zhang, and Bin Cui. 2020. Graph neural networks in recommender systems: a survey. *arXiv preprint arXiv:2011.02260* (2020).

[32] Shu Wu, Yuyuan Tang, Yanqiao Zhu, Liang Wang, Xing Xie, and Tieniu Tan. 2019. Session-based recommendation with graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 346–353.

[33] Inc. Xilinx. 2021. *AXI High Bandwidth Memory Controller v1.0 LogiCORE IP Product Guide.* https://www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf

[34] Inc. Xilinx. 2021. *Vitis High-Level Synthesis User Guide UG1399 (v2020.2).* https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_2/ug1399-vitis-hls.pdf

[35] Liqi Yang, Linhan Luo, Lifeng Xin, Xiaofeng Zhang, and Xinni Zhang. 2021. DAGNN: Demand-aware Graph Neural Networks for Session-based Recommendation. *arXiv preprint arXiv:2105.14428* (2021).

[36] Guorui Zhou, Kun Gai, Xiaoqiang Zhu, Chenru Song, Ying Fan, Han Zhu, Xiao Ma, Yanghui Yan, Junqi Jin, and Han Li. 2018. Deep Interest Network for Click-Through Rate Prediction. 1059–1068. https://doi.org/10.1145/3219819.3219823

[37] Guorui Zhou, Na Mou, Ying Fan, Qi Pi, Weijie Bian, Chang Zhou, Xiaoqiang Zhu, and Kun Gai. 2019. Deep interest evolution network for click-through rate prediction. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 5941–5948.