FAFNIR: Accelerating Sparse Gathering by Using Efficient Near-Memory Intelligent Reduction

Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Da Eun Shim, Sung-Kyu Lim, Hyesoon Kim
Georgia Institute of Technology, Atlanta, GA
{bahar.asgari, rhadidi, jcao62, daeun}@gatech.edu, limsk@ece.gatech.edu, hyesoon@cc.gatech.edu

***Abstract*—Memory-bound sparse gathering, caused by irregular random memory accesses, has become an obstacle in several on-demand applications such as embedding lookup in recommendation systems. To reduce the amount of data movement, and thereby better utilize memory bandwidth, previous studies have proposed near-data processing (NDP) solutions. The issue of prior work, however, is that they either minimize data movement effectively at the cost of limited memory parallelism or try to improve memory parallelism (up to a certain degree) but cannot successfully decrease data movement, as prior proposals rely on spatial locality (an optimistic assumption) to utilize NDP. More importantly, neither approach proposes a solution for *gathering* data from random memory addresses; rather they just offload operations to NDP. We propose an effective solution for *sparse gathering*, an efficient near-memory intelligent reduction (Fafnir) tree, the leaves of which are all the ranks in a memory system, and the nodes gradually apply reduction operations while data is gathered from *any* rank. By using such an overall tree, Fafnir does not rely on spatial locality; therefore, it minimizes data movement by performing *entire* operations at NDP and fully benefits from parallel memory accesses in parallel processing at NDP. Further, Fafnir offers other advantages such as using fewer connections (because of the tree topology), eliminating redundant memory accesses without using costly and less effective caching mechanisms, and being applicable to other domains of sparse problems such as scientific computing and graph analytics. To evaluate Fafnir, we implement it on an XCVU9P Xilinx FPGA and in 7 nm ASAP ASIC. Fafnir looks up the embedding tables up to 21.3× more quickly than the state-of-the-art NDP proposal. Furthermore, the generic architecture of Fafnir allows running classic sparse problems using the same 1.2 mm^2 hardware up to 4.6× more quickly than the state of the art.**

I. Introduction

The classic problem of irregular memory accesses in sparse problems, which was first identified as a major challenge by the high-performance computing (HPC) community [1], has recently become an obstacle in a wider range of applications. An important and growing domain of such applications is the recommendation systems broadly used in industry (e.g., Facebook, Netflix, Youtube, Spotify, Baidu, Alibaba) to suggest content such as music, products, and videos to users [2]–[5]. Recommendation systems include large embedding tables (sets of embedding vectors) consisting of users' data and features [3], [6]–[9]. The embedding tables are dense data structures. However, looking up the embedding table causes irregular *sparse gathering*, which has been determined to be the memory-bandwidth-hungry component in recommendation systems [6]–[9] and the main source of performance bottleneck.

Embedding lookup, on the one hand, demands high memory bandwidth, and on the other hand, consists of reduction operations with low compute intensity. These two features encouraged prior work to propose near-data processing (NDP) solutions, namely TensorDIMM [9] and RecNMP [8]. TensorDIMM [9] splits the embedding vectors across DIMMs to utilize rank-level parallelism for reading *individual* embedding vectors. Accordingly, it splits the reduction operations across the DIMMs. As a result, TensorDIMM successfully performs all reductions at DIMMs and minimizes data movement from memory to the cores by sending only outcome vectors rather than all embedding vectors. However, the downside of TensorDIMM is that it does not sufficiently utilize memory parallelism because it uses column-major order, which fundamentally breaks the row-buffer locality in the DRAM system.

As opposed to TensorDIMM [9], RecNMP [8] utilizes rank-level parallelism for reading *distinct* embedding vectors. Thus, the performance of RecNMP scales better as more ranks are added to the memory system. The main issue of RecNMP, however, is that it cannot fully utilize the NDP in the lack of spatial locality. In other words, if related embedding vectors do not reside in the same DIMM, RecNMP fails to apply reduction at NDP and therefore forwards them to the cores, which in turn does not sufficiently decrease data movement. This challenge is more pronounced in large embedding tables, in which the related embedding vectors are less likely to be located in the same DIMMs. Besides the aforementioned main challenges, the previous studies did not envision a few other important aspects in designing an NDP solution. For instance, RecNMP [8] uses costly but less effective caching mechanisms to reduce memory accesses. Further, neither of them considers the overhead of connections in larger systems. Finally, previous studies focus only on one application and do not show whether their solutions are applicable in other domains without hardware modifications.

An effective solution for *sparse gathering* must co-optimize data movement and parallelism. Our key insight to enable such a solution is to *process data while we gather them* rather than processing them where they reside, mainly because in sparse gathering, co-related data (i.e., operands of an operation) do not reside in a single location of memory. To this end, we propose an efficient near-memory intelligent reduction (Fafnir[1]), which makes four main contributions:

- *Low data movement and few connections:* Fafnir uses a tree that connects all ranks in a memory system, thereby enabling full reduction at NDP on embedding vectors from any rank. Such a tree not only guarantees data-movement reduction but also requires fewer connections than current solutions in larger systems.
- *High-level parallelism and scalability:* Fafnir utilizes rank-level parallelism in reading distinct embedding vectors, provides DIMM-level parallelism, and more importantly takes advantage of memory parallelism in utilizing available parallel processing at NDP. As a result, the performance of Fafnir scales as more ranks are added to the system or when batch size increases.
- *Effective memory-access reduction:* Fafnir reduces memory accesses by reading only the unique memory accesses (a fraction of all indices) in a batch of embedding lookups and using them through the reduction process in the tree as many times as required without using any caching mechanism that is less effective (low hit rate) and can cause a performance bottleneck.
- *Applicable to other sparse problems:* Fafnir is the first generic NDP solution that can execute sparse problems from various domains such as scientific computation and graph analytics without modifications in hardware. Genericity is key to reducing fabrication costs.

We implement Fafnir on an XCVU9P FPGA (utilizing up to 3% of the resources) and design a 1.25 mm^2 ASIC at 7 nm. Fafnir looks up the embedding tables up to 9.9×, 15.4×, and 21.3× more quickly than RecNMP for batch sizes of 8, 16, and 32, respectively, which is mainly achieved thanks to a tiny (i.e., 0.121 mm^2) chip between the memory channels and core. Additionally, Fafnir executes sparse problems from other application domains up to 4.6× more quickly than the state-of-the-art NDP accelerator [10].

II. Sparse Gathering

Sparse data and randomness in the nature of a problem are the two main sources of irregular memory accesses, resulting in the major problem of costly data movement [11]–[18]. The challenge of irregular memory accesses, first identified as a major challenge in the high-performance computing (HPC) community [1], is now an obstacle to a wider range of applications in which *randomness in the nature of a problem* contributes to creating irregular memory accesses. This section reviews recommendation systems, a growing application that captures sparse gathering.

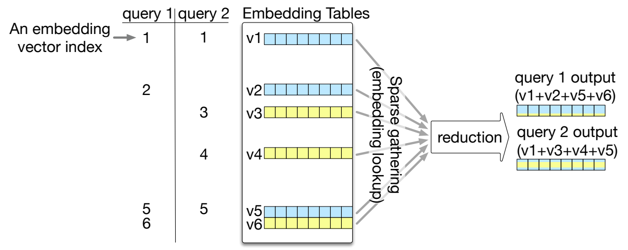

query 1
query 2
Embedding Tables
An embedding vector index
1
1
2
3
4
5
6
5
v1
v2
v3
v4
v5
v6
Sparse gathering (embedding lookup)
reduction
query 1 output (v1+v2+v5+v6)
query 2 output (v1+v3+v4+v5)

Figure 1: An example of embedding lookup (sparse gathering) and reduction for two queries.

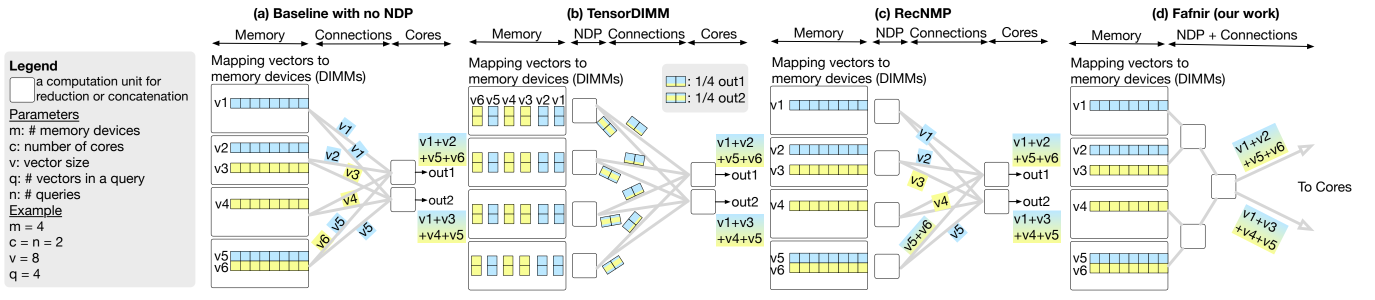
Recommendation Systems: To recommend contents such as music, video, and products to users, recommendation systems are broadly used throughout industry [2], [7], [19]. For instance, recommendation models consume 65% of artificial-intelligence inference cycles in the production data-center of Facebook [7]. The recommendation systems consist of (i) embedding tables, the sets of embedding vectors that contain users' data and features, followed by (ii) neural networks, including fully connected [3] and/or rectified-linear-unit [2] layers. To recommend content, first, the related embedding vectors are gathered from the embedding tables. Then, a simple reduction operation (e.g., element-wise summation, minimum, average) is applied on the gathered embedding vectors to derive a single vector, which is then sent to the neural networks for further processing. The focus of this paper is the first two steps: embedding lookup and performing reduction. While embedding tables are *dense*, looking them up causes random memory accesses, which results in *sparse* gathering. Therefore, embedding lookup is the memory-bandwidth-hungry part of the recommendation system. Besides, the embedding lookup and the reduction operations capture lower computation intensity and more cache misses compared to neural networks [7]. Such characteristics put recommendation systems in the memory-bound region of the roofline model of CPUs and far below the ceiling [20] because of memory bandwidth underutilization.

Terminology: Figure 1 summarizes the terminology used in this paper. An embedding vector is identified by an **index**, a set of which creates a **query**. Reading data from irregular random locations of memory, indicated by a query, is **sparse gathering**. An example of sparse gathering in recommendation systems is **embedding lookup**. The outcome of an embedding lookup (two or more embedding vectors) is reduced into one vector by applying a **reduction** such as element-wise summation [20].

III. Challenges

Despite common efforts to reduce the embedding vector dimension [21] or number of embedding vectors [20], embedding tables occupy multiple gigabytes of memory. Such constraints necessitate the distribution of the embedding tables across multiple memory devices to satisfy memory capacity requirements [3]. Sparse gathering from random addresses scattered over a large memory system requires maximizing memory-bandwidth utilization. However, the *processor-centric* organization of CPUs and GPUs and the reused-optimized structure of their memory hierarchy,

[1]Fafnir (/faːvnər/) is a star in the constellation of Draco.



Section		(a) General	(a) This example	(b) General	(b) This example	(c) General	(c) This example	(d) General	(d) This example
III.A,C	Transferred data (from memory/NDP to cores)	n x q x v	2 x 4 x 8 = 64	n x v	2 x 8 = 16	min: n x v max: n x q x v	6 x 8 = 48 (counting v1 once)	n x v	2 x 8 = 16
III.B	Reading different vectors Reading a vector	parallel ranks sequential columns		random rows parallel ranks		parallel ranks sequential columns		parallel ranks sequential columns	
III.B	Parallel compute at NDP	N/A	N/A	v	8	n x (q-1) x v (in theory)	2 x (4-1) x 8 = 48	n x (q-1) x v	2 x (4-1) x 8 = 48
III.B,C	Scalar operations | NDP	0	0	n x (q-1) x v	2 x (4-1) x 8 = 48	min:0 / max: n x (q-1) x v	1 x 8 = 8	n x (q-1) x v	2 x (4-1) x 8 = 48
	Scalar operations | cores	n x (q-1) x v	2 x (4-1) x 8 = 48	(m-1) x n concat.	(4-1) x 2 = 6 concat.	min: 0 / max: n x (q-1) x v	5 x 8 = 40	0	0
III.C	DIMM-level parallelism	No		No		No		Yes	
III.D	#Connections (excluding connections to memory)	c x m	2 x 4 = 8	c x m	2 x 4 = 8	c x m	2 x 4 = 8	(2m - 2) + c	(2 x 4 - 2) + 2 = 8

Figure 2: Comparing NDP-based solutions for embedding lookup: (a) Baseline with no NDP, (b) TensorDIMM [9], (c) RecNMP [8], and (d) Our proposed solution, Fafnir.

conspire against efficient and fast sparse gathering. As a results, prior studies proposed TensorDIMM [9], RecNMP [8], and Centaur [22], *data-centric* solutions to process data (i.e., the embedding vectors) where they reside (i.e., in DIMM and/or rank). Despite the advantages of these NDP solutions, they left unsolved a few key challenges of *sparse gathering*. In the following, we elaborate on different challenges, solving some of which has been the target of a prior study that in turn causes the other challenges. Figure 11 in Section VI provides detailed discussions on the challenges using quantitative results.

A. Data Movement

The large amount of data movement has been the main concern of sparse gathering. For instance, as Figure 2a shows, to apply two non-compute-intensive reduction operations on six embedding vectors, the vectors must be transferred all the way to cores. For example, vector 5 (v5) is required by two queries; thus, it is transferred twice. In general, to perform n queries of size q on vectors including v elements, $n \times q \times v$ elements must be transferred from the memory system to cores. To solve this challenge, TensorDIMM [9] performs reductions in the DIMMs and transfers only the results to the cores (Figure 2b). As a result, instead of transferring all q vectors in a query, it sends only one vector, hence reducing the amount of data movement q times to $n \times v$. TensorDIMM splits the embedding vectors and distributes them over DIMMs and creates $1/q$ of each output vector at a DIMM. Then, the cores just concatenate the partitions of the outputs.

B. Lack of Utilization of Row Buffer Locality

Even though TensorDIMM effectively reduces data movement, it is not as effective in sufficiently utilizing row-buffer hits because it uses column-major order, which fundamentally breaks the row-buffer locality in the DRAM system. More specifically, while TensorDIMM can utilize rank-level parallelism to read the elements of individual embedding vectors, split over different DIMMs, it must access random rows to read *distinct* embedding vectors in a query (e.g. vectors of query 1 including v1, v2, v5, and v6). Accordingly, only v scalar operations can be performed in *parallel* at NDP. Although TensorDIMM performs all $n \times (q-1) \times v$ operations at NDP, only v of them are processed in *parallel*, while the rest can be *pipelined*. For instance, for query 1, all DIMMs do the following subsequently: read their own part of v1 from a row (but not necessarily reading the entire row buffer), then read v2 from another row, do a partial sum of size v/m (v1+v2), simultaneously access another row to read v5, add it to the partial sum while reading v6 from another row. This approach particularly disturbs achieving low latency.

Splitting embedding vectors across more ranks causes poor utilization of row-buffers (i.e., we must open a row, but read a *smaller* fraction of it). To improve parallel computation at NDP, RecNMP [8], another NDP solution for embedding lookup, distributes embedding vectors across the ranks, as shown in Figure 2c. In this approach, reading *distinct* embedding vectors utilizes rank-level parallelism, while elements of each vector are read from sequential columns. As a result, RecNMP can more effectively increase rank-level parallelism by adding more ranks to the system. However, the downside of this approach is that even though in theory entire operations for all queries (i.e., $n \times (q-1) \times v$) can be performed in parallel at NDP, RecNMP might not achieve it because of imperfect spatial locality. below.

C. Relying on Spatial Locality

Although RecNMP utilizes rank-level parallelism in reading distinct embedding vectors, it does not guarantee processing them *all* at NDP, mainly because it does not provide DIMM-level parallelism. In many real-world applications, embedding vectors of a query are scattered over many random DIMMs, where DIMM-level parallelism (i.e., channel-level reduction) is essential. For instance, based on the birthday paradox, the probability of having a query with indices on the same channel is only up to 25% in a four-channel system. Consequently, as in many cases, the raw data needs to be transferred to the cores, so memory bandwidth may not be fully utilized. As a result, even though under perfect circumstances, maximum $n \times (q-1) \times v$ operations can be done at NDP, in the worst case, all of them might need to be done at the cores. For instance, in Figure 2c, only two embedding vectors (i.e., v5 and v6) are reduced at NDP, and others are forwarded to the cores. Relying on spatial locality has two other consequences. First, increasing batch size does not necessarily result in more utilization of parallel computation at NDP, hence achieving higher throughput. Second, while in the perfect scenario only n output vectors (i.e., $n \times v$ elements) are transferred from the memory to the cores, in the worst case, all $n \times q \times v$ elements must be transferred. Therefore, reducing data movement is also not guaranteed.

D. Connection Overhead

The other challenge of implementing embedding lookup is the overhead of connections. As the embedding tables are often large, they necessitate model parallelism (i.e., splitting and distributing tables across memory devices), as shown in Figure 2a. On the other hand, the neural-network layers of the embedding systems are small enough to utilize data parallelism (i.e., implementing copies of a neural network on different computing devices). The combination of model parallelism for embedding tables and data parallelism for neural networks in recommendation systems requires costly all-to-all connections [3] (e.g., Figure 2a) between the memory devices and computing devices (e.g., CPU or GPU cores) so that embedding vectors can be gathered from any memory device and be forwarded to any computing device. The aforementioned previous studies have not proposed any solutions to reduce the number of connections. Therefore, similar to the baseline, they all require $c \times m$ connections to implement all-to-all communication, which is not only costly but also limits the scalability. To accelerate sparse gathering and prevent the communication from becoming a bottleneck, Centaur [22] uses high-bandwidth communication links and then applies the reduction operations in a separate unit. Thus, unlike TensorDIMM, Centaur does not reduce data movement but instead transfers data more quickly.

E. Using Caches to Reduce Memory Accesses

The last challenge of implementing NDP solutions for embedding lookup is eliminating extra memory accesses. Observations suggest that a batch of queries have common embedding vectors. Thus, not all the memory accesses corresponding to every single embedding vector are necessary. For instance, in the example of Figure 1 and Figure 2, both query 1 and query 2 require embedding vector 5 (v5). RecNMP [8] proposes using caches at NDP. Caching, however, is not the most effective solution, as no more than a 50% hit rate can be achieved [8]. Even achieving such a hit rate requires a 128 KB cache that adds extra hardware overhead (e.g., 38% area [8]). Besides, the cache accesses can potentially cause a performance bottleneck.

IV. Fafnir

This section proposes our solution to address the challenges, listed in Section III. Our goal is to provide an all-in-one solution to reduce data movement and provide memory and computation parallelism without relying on spatial locality. To achieve this goal, our key insight is to *process data while it is gathered* rather than processing data where it resides, mainly because in sparse gathering, data (i.e., different embedding vectors) do not reside in a single memory location; rather it is scattered. Based on this insight, we propose Fafnir, a data-centric solution for embedding lookup that, unlike prior data-centric solutions, gradually applies the reduction on data while gathering them from random memory devices. The overhead for achieving the benefits explained in the following is $m-1$ processing elements compared to prior work (that is, adding a $0.121\,mm^2$ at $7\,nm$ chip as shown in Section VI).

A. Main Contributions

Using an Overall Tree Structure: To enable applying reduction on embedding vectors from *any* memory devices without relying on spatial locality, we use an overall reduction tree, the leaves of which are connected to the ranks of a memory system and the nodes are reduction engines (Figure 2d). In this way, we guarantee that all embedding vectors in a query are definitely reduced within the tree at NDP – it could occur in a leaf if the embedding vectors are from neighboring memory devices or could occur at least at the root if the vectors are at the remotest locations. Therefore, while in all three schemes (i.e., TensorDIMM, RecNMP, and Fafnir) the mapping of vectors to DIMMs equally define the load of each NDP, only in Fafnir are the entire operations done at NDP, regardless of the mapping of vectors to DIMMs. Since Fafnir performs all the reduction operations at NDP, it guarantees decreasing data movement. In other words, similar to TensorDIMM, only the $n \times v$ elements corresponding to the outputs are transferred from the NDP to the cores. The tree structure of Fafnir also optimizes the number of connections. More specifically, instead of connecting NDP to the cores through the costly $c \times m$ all-to-all connections, Fafnir integrates computations within $2m-2$ connections and then forwards them to the cores through c connections (i.e., total $(2m-2)+c$, Figure 2d). As a result of the fewer connections when adding more computation devices, Fafnir is also more scalable compared to prior proposals.

Parallelizing Memory Accesses & Computations: To fully utilize the tree and thus provides parallel computation while also reading data in parallel, Fafnir simultaneously activates distinct routes of the tree from arbitrary leaves to the root to process a batch of queries. Fafnir flows data corresponding to distinct queries through the tree in such a way that they do not conflict and hence their latency does not affect one another. As a result of this mechanism, Fafnir guarantees full utilization of the parallel computation at NDP (i.e., $n \times (q-1) \times v$). Therefore, not only by *adding more ranks* to the system, but also by *increasing the batch size* (processing more queries), we can better utilize the parallelism and achieve higher throughput.

No Caching Mechanisms: Fafnir uses a novel approach for processing a batch of queries with shared indices that does not require caching mechanisms for eliminating redundant accesses to memory. In other words, Fafnir reads only the unique indices from memory and then uses them as many times as required without storing data in a cache, hence preventing overhead such as searching, reading from, and writing to a cache. Fafnir rearranges a batch of queries and treats them as a set of unique indices. Therefore, Fafnir accesses each unique index only once, and then, based on the query indices, it reduces the corresponding indices within the tree. Our observations, shown in Figure 3, illustrate the opportunity to effectively benefit from our novel batch processing mechanism (details in Section IV-C). This mechanism of Fafnir also improves energy efficiency.

Executing Various Sparse Problems: Customized hardware has not often been selected as a viable option. Instead, general-purpose hardware has usually been used for executing applications such as sparse problems, even though their performance is dramatically low. A reason for this is the economic aspect. Extensive customization has been expensive for narrow applications, even if such hardware offers significant performance benefits. To deal with the cost challenge, custom hardware solutions must be generic and applicable to a reasonable range of applications. To this end, we envision hardware for Fafnir that is generic enough to be used for executing other sparse applications, that include graph algorithms and scientific computations including matrix algebra, the main kernel of which is sparse matrix-vector multiplication (SpMV).

B. The Top-Down Overview of Fafnir

Software Support: Fafnir is a DDR-based NDP connected to a host for the software support. The host is responsible for mapping data to the memory addresses, compiling the NDP kernels into a set of memory accesses, and calling Fafnir for executing NDP kernels by transmitting memory access requests to the root of the tree. The type of memory accesses differs based on the program. For instance, for embedding lookup, the host sends batches of memory read addresses, whereas for SpMV (see Section IV-D), it forwards stream accesses from all occupied ranks by specifying the initial memory address and the size of the stream. Besides, while for an embedding-lookup kernel the application software at host arranges the queries, it performs vectorization for SpMV. The distribution of the memory accesses across the memory devices (ranks) is a result of the program behavior. The root receives the requests in the form of regular DDR4-compatible command/address (C/A) signals, decodes them, and forwards them to corresponding (all if needed) DIMM/ranks across all parallel ranks. Ranks read data through DDR4-compatible data (DQ) signals and then all the special steps of Fafnir to gradually apply reduction operations from leaves to the root occur. Finally, the root sends the outcome back to the host.

Architecture: Figure 4a shows an overview of the Fafnir architecture, consisting of 32 ranks, and hence 31 processing elements (PEs), connected in a tree structure. In current implementation of Fafnir, one leaf PE is connected to two ranks (i.e., 1PE:2R) and concurrently accesses them without creating conflicts by using the same techniques used in prior work [8], [9]. Similarly, depending on system requirements, other scales (e.g., 1PE:4R or 1PE:1R) are implementable. The PEs can be grouped as nodes in various ways. Each node would be a sub-tree of PEs, implemented in FPGA or ASIC. For instance, we can fabricate one PE chip of size $274\mu m \times 282\mu m$ at 7 nm (Figure 4a left layout) and embed it in a DIMM or put seven PEs together in a single $492\mu m \times 575\mu m$ chip to connect all the four DIMMs in a channel. In this paper, we implement two types of nodes: DIMM/rank and channel nodes. Accordingly, the Fafnir configuration consists of four DIMM/rank and one channel node. The nodes borrow their names from the source of their inputs. The input to each DIMM/rank node is from eight ranks (4 DIMMs, 2 ranks per each). A DIMM/rank node has seven PEs. Likewise, the channel node has three PEs and its inputs come from four channels.

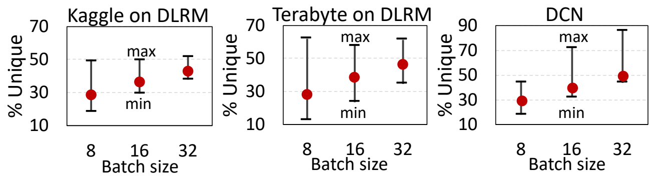


Figure 3: The percentage of unique indices in batches of queries (see model description in Section V.)

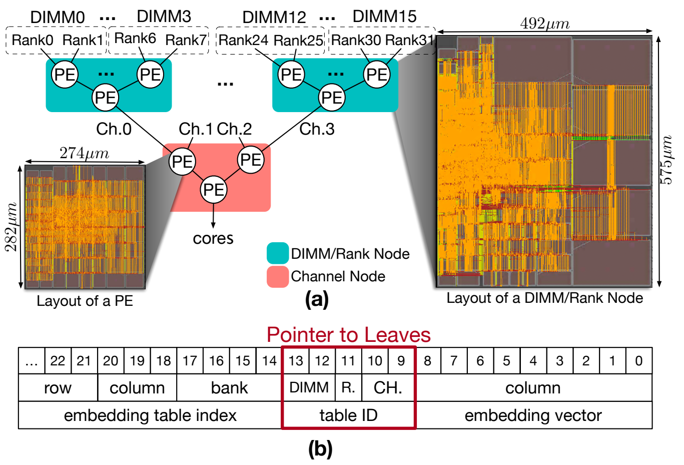


Figure 4: (a) The architecture of Fafnir tree, consisting of DIMM/rank and channel nodes and ASIC designs at 7 nm for a PE and a DIMM/rank node. (b) The mapping of embedding tables to memory addresses.

Figure 4b illustrates the mapping of embedding tables to 32 ranks of our target memory system – we map embedding vectors (e.g., each 512 bytes) to distinct ranks. Data flowing from leaves to the root of the tree includes a header and a value (the gathered data). The header consists of two fields: *indices* and *queries*. The *indices* indicate the locations of memory from which data have been gathered (i.e, the bits [9-13] shown in Figure 4b). The *queries* indicate a list of indices for different queries that have not been visited, yet. For instance, assume that we have a query with indices `1, 2, 5, 6`, and data in a PE is the result of reducing data from indices `1, 2` and is yet to be reduced with data from indices `5, 6`. As a result, the output of that PE will have a header of `[indices:1,2|queries:5,6]`. By approaching the root and visiting more PEs of the tree, the indices from the *queries* field of the header are shifted to the *indices* field. Once data arrives at the root of the tree, the *queries* field will be empty, and the *indices* field will indicate a complete set of reduced indices for that query.

Microarchitecture of PEs: Figure 5 shows the microarchitecture of a PE, including two inputs (A and B) coming from a rank or the upstream PE in the tree architecture, and one output going to the downstream PE. A PE consists of two input FIFO buffers connected to compute units, the outputs of which are merged and directed to the output through a merge unit. The task of each PE is to process the headers and decide whether to reduce the inputs and assign a new header to it, or just forward them as they are. To enable processing a batch of inputs, we instantiate compute units, each of which iteratively compares one element of an input (e.g., `B[x]`) with all elements of the other input (e.g., `A`). More specifically, the entire *queries* field of `B[x]` is compared with the *indices* field of `A[i]` (i.e., `B[x].queries[j]` and `A[i].indices` are compared). If `B[x].queries[j]` contains all elements of `A[i].indices`, the compute unit performs a reduction. If none of them match, it forwards `B[x]`. In each PE, we also compare the two inputs in the other way to make sure that the *queries* field of `A[i]` is also matched with the *indices* field of `B[x]`. Since we process the inputs of a PE in parallel, the compute units may generate the exact same outputs concurrently, or multiple compute units may generate multiple outputs, the data of which are equal. In the first case, the redundant outputs must be removed, and in the second case, the outputs with the same data must be merged and the *queries* field in their headers must be merged (i.e., concatenated). Such post-processing is the task of the merge unit.

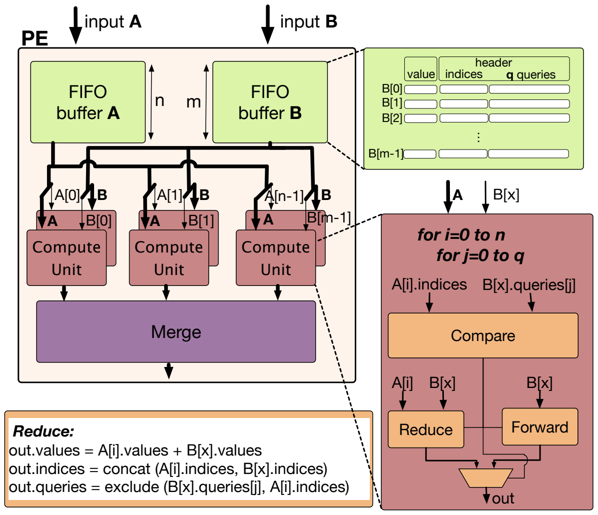


Figure 5: The microarchitecture of a PE including FIFO buffers, compute, and merge units, showing the data path from leaves to the root.

All PEs across the tree are identical. The size of the PE (the size of input buffers A and B and the number of compute units) could be tailored to better handle different batch sizes. We define sizing based on the maximum size of inputs. Since we are processing batches of queries, each combination of the two inputs of a PE might be required by one of the queries in the batch. Therefore, in the worst case, a PE will need to generate all the possible combinations of its input to the output, which is a maximum of three combinations: Each of the inputs can individually be forwarded to output, or they can be reduced. Therefore, in theory, the number of outputs of a PE with two inputs of sizes n and m is $nm + n + m$.

The output size of a PE defines the input size of the downstream PE. Therefore, as we move closer to the root, the size of the outputs and hence the size of consecutive inputs is supposed to be increasing, which demands larger buffers and more compute units. However, in fact, *the number of outputs of each PE is limited by the batch size.* This is simply because not all the combinations of the inputs are being used by a limited number of queries. While hardware is fixed for a batch size, larger batch sizes defined by software in various application domains are served as several small batches at hardware. Therefore, the maximum number of outputs for a PE is calculated as $min(nm + n + m, B)$, in which B is the batch size. Table I lists the total size of buffers for PEs and nodes, which is the same for PEs at any level of the tree for three batch sizes. As Figure 5 shows, the buffers contain $n = m$ entries, each including a 512 B value and a 10 B header ($16 \times 5/8$) for $q = 16$ (i.e., each query includes maximum 16 indices) and 5-bit *indices*/*queries* fields for identifying embedding vectors from 32 embedding tables. In our configurations, $n = m = B$ also defines *the number of compute units* in a PE. When the size of inputs is smaller than the number of compute units, some compute units will simply have no value and remain idle.

Table I: FIFO buffer sizes that are sum of all buffers in all PEs (B is batch size).

Node	PE buffer (KB)			Node buffer (KB)		
	B = 8	B = 16	B = 32	B = 8	B = 16	B = 32
DIMM/Rank Channel	4.6	9.3	18.5	32.4 13.9	64.8 27.8	129.5 55.5

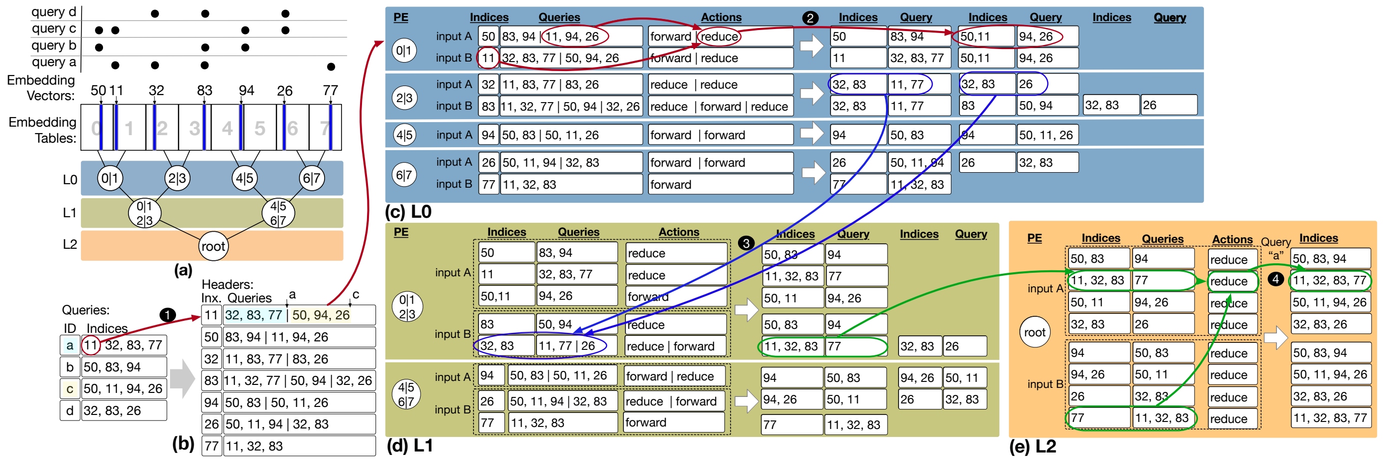

query d
query c
query b
query a
Embedding Vectors:
50 11 32 83 94 26 77
Embedding Tables:
0 1 2 3 4 5 6 7
L0
L1
L2
0|1
2|3
4|5
6|7
0|1 2|3
4|5 6|7
root
(a)
Queries:
ID Indices
a 11, 32, 83, 77
b 50, 83, 94
c 50, 11, 94, 26
d 32, 83, 26
Headers:
Inx. Queries
a
c
11 | 32, 83, 77 | 50, 94, 26
50 | 83, 94 | 11, 94, 26
32 | 11, 83, 77 | 83, 26
83 | 11, 32, 77 | 50, 94 | 32, 26
94 | 50, 83 | 50, 11, 26
26 | 50, 11, 94 | 32, 83
77 | 11, 32, 83
(b)
PE Indices Queries Actions Indices Query Indices Query Indices Query
0|1 input A 50 | 83, 94 | 11, 94, 26 | forward | reduce | 50 | 83, 94 | 50,11 | 94, 26
input B 11 | 32, 83, 77 | 50, 94, 26 | forward | reduce | 11 | 32, 83, 77 | 50,11 | 94, 26
2|3 input A 32 | 11, 83, 77 | 83, 26 | reduce | reduce | 32, 83 | 11, 77 | 32, 83 | 26
input B 83 | 11, 32, 77 | 50, 94 | 32, 26 | reduce | forward | reduce | 32, 83 | 11, 77 | 83 | 50, 94 | 32, 83 | 26
4|5 input A 94 | 50, 83 | 50, 11, 26 | forward | forward | 94 | 50, 83 | 94 | 50, 11, 26
6|7 input A 26 | 50, 11, 94 | 32, 83 | forward | forward | 26 | 50, 11, 94 | 26 | 32, 83
input B 77 | 11, 32, 83 | forward | 77 | 11, 32, 83
(c) L0
PE Indices Queries Actions Indices Query Indices Query
0|1 2|3 input A 50 | 83, 94 | reduce; 11 | 32, 83, 77 | reduce; 50,11 | 94, 26 | forward
input B 83 | 50, 94 | reduce; 32, 83 | 11, 77 | 26 | reduce | forward
50, 83 | 94; 11, 32, 83 | 77; 50, 11 | 94, 26; 50, 83 | 94; 11, 32, 83 | 77; 32, 83 | 26
4|5 6|7 input A 94 | 50, 83 | 50, 11, 26 | forward | reduce
input B 26 | 50, 11, 94 | 32, 83 | reduce | forward; 77 | 11, 32, 83 | forward
94 | 50, 83; 94, 26 | 50, 11; 77 | 11, 32, 83; 94, 26 | 50, 11; 26 | 32, 83
(d) L1
PE Indices Queries Actions Query "a" Indices
root input A 50, 83 | 94 | reduce; 11, 32, 83 | 77 | reduce; 50, 11 | 94, 26 | reduce; 32, 83 | 26 | reduce
input B 94 | 50, 83 | reduce; 94, 26 | 50, 11 | reduce; 26 | 32, 83 | reduce; 77 | 11, 32, 83 | reduce
50, 83, 94; 11, 32, 83, 77; 50, 11, 94, 26; 32, 83, 26; 50, 83, 94; 50, 11, 94, 26; 32, 83, 26; 11, 32, 83, 77
(e) L2
❶ ❷ ❸ ❹

Figure 6: Concurrent batch processing and eliminating redundant memory accesses in Fafnir: (a) A batch of four queries that access random embedding vectors from eight embedding tables and a three-level Fafnir tree (b) Extracting the unique indices of four queries and creating the headers of requests to be forwarded to Fafnir. The steps of processing the four queries through the PEs at three levels of tree: (c) L0, (d) L1, and (e) L2.

C. Key Mechanisms

This section describes how Fafnir performs *eliminates redundant memory accesses* and *concurrent batch processing*. This mechanism, however, does not rely on the batch processing. For instance the same mechanism can also be used for interactive processing, in which all nodes would either forward or reduce without performing any comparisons. Fafnir first reads data corresponding to only the unique indices once, and then uses them within different queries as many times as required without using any caching techniques. The example in Figure 6 shows the steps of batch processing for a batch of four queries (i.e., a, b, c, and d) that access random embedding vectors from eight embedding tables and then processing them through a three-level tree, shown in Figure 6a. Only in this example, we assume that the indices to embedding vectors are created by concatenating the index within a table with a table number (e.g., `50` indicates index `5` from table `0`).

To decrease the number of memory accesses, the host extracts the unique indices used in a batch of queries and creates the headers including *indices* (i.e., Inx) and *queries* fields (Figure 6b). To do so, the unique indices are added to the *indices* field. Then, all the indices of the queries, including that unique index but excluding the unique one, are added to the *queries* field. For instance, for the unique index `11` ❶, we add the following to the *queries* field: ~~`11`~~, `32, 83, 77` from query a and `50,` ~~`11`~~`, 94, 26` from query c (`11` is excluded from both). In this way, instead of a total of 14 memory accesses, we access seven unique ones: `50, 11, 32, 83, 94, 26, 77`. The tables in Figures 6c, 6d, and 6e list the details of the processing steps at levels L0, L1, and L2 of the tree, respectively. These tables list (i) the headers of the input A and input B to each PE, (ii) the actions taken based on each comparison – each action corresponds to the result of comparisons of one item in the *queries* header, (iii) the header of raw outputs of each PE before merging, and (iv) the inputs to the next PEs, which are basically the merged outputs of the previous PEs.

PE `(0|1)` (similar to others) has two inputs, A and B. As the *queries* fields of A and B indicate, data from indices `50` and `11` will be used in two queries. In `(0|1)`, a compute unit compares item `[83,94]` of A with the index of B (i.e., `11`) and since `11` is not included in `[83,94]`, the compute unit forwards the value coming from input A, with its initial header of `[indices:50|queries:83,94]`. Likewise, item `[11, 94, 26]` of A is compared with the index of B (i.e., `11`) and finds a match, thus reducing the values of A and B and creating the new header of `[indices:50,11|queries:94, 26]` ❷. The *indices* field of the header is created by concatenating the indices of A and B and the *queries* field is created by excluding the indices of A and B from `[11, 94, 26]`. The compute units in PE `(0|1)` do the same for items `[32, 83, 77]` and `[50, 94, 26]` of input B, resulting in a forward and a reduce.

As Figure 6c shows, the initial outputs of PE `(0|1)` include the header `[indices:50,11|queries:94,26]` twice. In such a case, the merge unit is responsible for eliminating redundant outputs. The three unique outputs of PE `(0|1)` create the input A of PE `(0|1|2|3)`, the input B of which includes two items that have been created similarly in PE `(2|3)`. The number of initial outputs of PE `(2|3)`, however, is five. Besides the redundant outputs with headers `[indices:32,83|queries:11 ,77]` and `[indices:32,83 |queries:26]`, PE `(2|3)` includes two groups of outputs with the same indices `32, 83`, but different *queries* field. In such a case, the headers must be merged, because they are two headers for one unique value. The result of such merging is a value with the header of `[indices:32,83 | queries:11,77|26]` (shown in Figure 6d), which goes to input B of PE `(0|1|2|3)` ❸. As the figure shows, because of merging, the size of input A and B never exceeds the batch size (i.e., four). The process of applying different actions on the inputs is similar in PE `(0|1|2|3)`, whereas

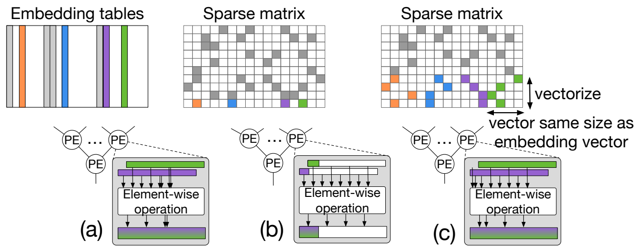


Figure 7: (a) Embedding lookup in Fafnir, (b) Using Fafnir for an SpMV with no mechanisms, and (c) Using vectorization to fully utilize Fafnir for SpMV.

here, each item of the *queries* field must be compared with the *indices* field of all items in the other input. Besides, as Figure 6c shows, in some cases, such as in PE (4|5), only one of the inputs exists, which automatically leads to a forward action. By iteratively processing data and gradually reducing them (when required) through the tree, we reach the root PE, the outputs of which indicate the initial queries (Figure 6e). For instance, the green lines (❹) show the final steps for creating query a.

D. Adapting Fafnir to SpMV

Sparse gathering is the *common* operation SpMV and embedding lookup, both of which can be implemented using a reduction tree. While this common feature allows adapting Fafnir to SpMV, maximizing the benefits for both requires addressing unique challenges that arise from their *differences*. The main difference between the *reduction* in embedding lookup and that in SpMV is that in embedding lookup, we reduce distinct vectors into one vector, whereas in SpMV, we need to reduce the elements of a vector into one element. Therefore, as Figure 7a illustrates, for an embedding lookup, each PE of Fafnir applies an element-wise reduction on two (or more) vectors and generates one output vector. For an SpMV, it is just the opposite: we need a reduction tree to sum the elements of a vector. As a result, the challenge is that if we simply use the reduction tree of Fafnir to execute SpMV, only one compute unit (reduce) of a PE will be utilized, as shown in Figure 7b. Our key insight to resolve this challenge is to use a *vectorization* technique along with an appropriate compression format. Figure 7c illustrates vectorization, in which each PE processes a vector of independent elements of the sparse matrix and separately applies the reduction operation on them. Vector size could be the same as embedding-vector size.

Because of their differences, embedding lookup and SpMV use different mechanisms on the same hardware. However, if the primary application of Fafnir would be SpMV, the control logic shown in Figure 5 would be

Table II: SpMV vs. embedding lookup

	SpMV	Embedding lookup
Indices	Unknown	Known
Memory-access type	Stream data and indices	Stream data only
Leaf PE	Multiplication with vector	Skip multiplication

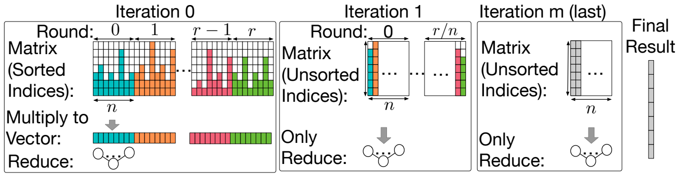


Figure 8: The iterations and rounds for SpMV on large sparse matrices using Fafnir when only n columns of the matrix fits to Fafnir at a time.

simpler because in SpMV, q is one and the iterations over q in compute units would not be necessary. Table II compares the mechanism of Fafnir for executing SpMV and embedding lookup. Unlike embedding lookup, for SpMV, the irregularity in memory accesses stems from *sparse data*. Because of such a difference, Fafnir handles memory accesses differently. First, as the second column of Table II lists, for SpMV, we do not know where the non-zero values of the sparse matrix are located. Therefore, when we read data from memory, the indices of the elements to be reduced are *unknown*. In fact, indices themselves are being read from memory. As a result, for SpMV, we stream both data and indices through the tree. Then, based on the indices, the tree reduces related values. In contrast, for embedding lookup, we know which indices we need to access. Therefore, we only stream data. The other difference between SpMV and embedding lookup is that the leaf PEs for SpMV first multiply data with the vector operands. The leaf PEs skip the multiplication for embedding lookup. Similar to embedding lookup, SpMV rather than caching mechanisms, uses a simple buffering, in which a vector operand is buffered in the multipliers until it is multiplied by the entire matrix operand.

To facilitate streaming sparse matrices, we suggest using the list-of-list (LIL) [23] compression format, which has become popular in recent sparse studies [24]–[26] (sometimes called other names such as linked list). Further, LIL is supported by the SciPy library [23], which makes its application more straightforward. LIL compresses the non-zero values of the original sparse matrix in one dimension and saves the indices corresponding to the other dimension of the matrix. As LIL compresses matrices only in one dimension, it facilitates splitting large matrices into chunks through their non-compressed dimension, hence facilitating parallel streaming. The ease of splitting and parallel

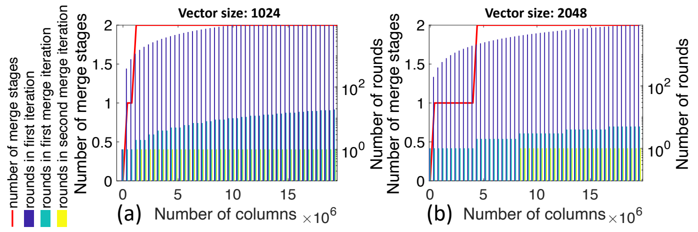


Figure 9: The number of iterations, rounds per iteration, and required merges for matrices with up to 20 million columns, for vector sizes (a) 1024 and (b) 2048. In our configuration for SpMV, vector size (i.e., the number of columns that fit in Fafnir tree) is 2048.

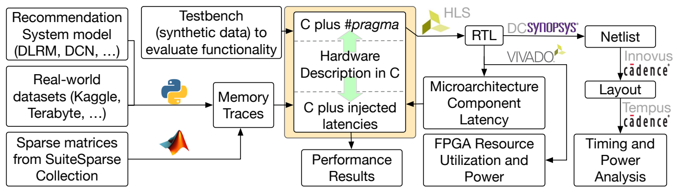

Recommendation System model (DLRM, DCN, ...)
Testbench (synthetic data) to evaluate functionality
C plus #pragma
Hardware Description in C
C plus injected latencies
HLS
RTL
DC SYNOPSYS
Netlist
VIVADO
Innovus cādence
Layout
Tempus cādence
Real-world datasets (Kaggle, Terabyte, ...)
Memory Traces
Microarchitecture Component Latency
Sparse matrices from SuiteSparse Collection
Performance Results
FPGA Resource Utilization and Power
Timing and Power Analysis

Figure 10: Experimental setup.

streaming is important in large sparse matrices (e.g., graph problems or HPC). To apply SpMV on large matrices that do not fit into Fafnir, we split them through their non-compressed dimension. Similar splitting is also used in the state-of-the-art NDP approach for SpMV [10].

As Figure 8 shows, we perform an SpMV in iterations, each consisting of several rounds. In the first iteration (iteration 0, which is functionally equivalent to the first step of Two-Step algorithm [10]), the matrix is multiplied by the vector operand, whereas all other iterations only merge the results of the previous iteration. We use the same hardware (DIMM/rank and channel nodes of Fafnir) for both types of iterations. During merge iterations (i.e., iterations > 0), leaf PEs skip the multiplications as they do in embedding lookup. In addition, during the merge iterations, the row indices are no longer sorted, but this does not impact the functionality of Fafnir. Figure 9 illustrates the number of required iterations and rounds per iterations for two vector sizes (i.e., 1024 and 2048) when the number of columns (and rows) increases up to 20 million. As the figure suggests, even for matrices with more than 5 million columns, no more than two merge stages are required.

V. Experimental Setup & Configuration

Figure 10 shows an overview of our design, implementation, and evaluation flow. We implement the microarchitecture of Fafnir (and the baselines) in C++. We use our hardware description in C++ for (i) RTL generation and subsequently FPGA and ASIC implementation, and (ii) performance evaluation. To generate RTL (in Verilog), we use related `#pragmas` as hints to describe the microarchitectures. We use Vivado HLS to generate RTL and Vivado to synthesize and implement our design on an XCVU9P FPGA, targeting a VCU1525 acceleration development kit, which includes four 16 GB DDR4 DIMMs (64 GB total per DIMM/rank node). Besides reporting resource utilization and power consumption for FPGA, we implement the ASIC design of Fafnir using the toolchain of Synopsys design compiler (DC), Cadence Innovus, and Cadence Tempus. As an input to our ASIC design, we use our same Verilog code generated by HLS and just substitute the BRAM blocks with memory cells. Our ASIC design is based on an Arizona State Predictive PDK (ASAP) 7nm technology node [28], a free PDK for non-commercial academic use. All performance numbers reported in this paper are based on FPGA-based C/RTL co-simulation results (as shown in Figure 10). For verifying functionality at scale we perform regression testing using large synthetic data through a C++ testbench for C/RTL co-simulation. To facilitate performance evaluation for large real-world data, we inject the FPGA post-implementation timing analysis @200MHz into our C++ emulator, the core description of which is initially used for RTL generation.

Table III: Sparse matrices from SuiteSparse [27].

ID	Name	Dim.(M)[1]	Density (%)	Application
RE	N_reactome	0.016	0.025	Biochemical
RI	rail582	0.056	1.2	Linear Prog.
HC	hcircuit	0.1	0.004	Circuit Sim.
2C	2cubes_sphere	0.101	0.016	Electromagnetic
TH	thermomech_dK	0.2	0.006	Thermal
FR	Freescale2	2.9	0.0001	Circuit Sim.
AM	amazon0601	0.4	0.002	Dir. Graph
WG	web-Google	0.91	0.0006	Dir. Graph
RO	roadNet-TX	1.3	0.0001	Unidir. Graph
KR	kron_g500-logn21	2	0.004	Unidir. Multiraph
WI	wikipedia-20070206	3.5	0.0003	Dir. Graph
LJ	soc-LiveJournal1	4.8	0.0002	Dir. Graph

[1] Dim.: dimension or the number of columns/rows of a square matrix.

We evaluate two applications: (i) recommendation systems including embedding lookup and (ii) graph analytics and scientific applications, both including SpMV. For scientific applications, we execute a matrix inversion algorithm (the most bottleneck-prone algorithm) using the lower-upper technique, which also iteratively calls SpMV. The inputs to our C++-based emulator are memory traces based on accesses to embedding tables of recommendation systems and the sparse matrices for SpMV-based applications. For the recommendation systems, we run Deep & Cross Network (DCN) [5] as well as Deep Learning Recommendation Models (DLRM) [3] based on two real-world open-source data sets: (i) the Criteo Ad Kaggle data set [29] containing approximately 45 million samples over seven days and (ii) the Criteo Ad Terabyte data set [30] sampled over 24 days. We logged the indices of embedding-table accesses and preprocess them using Python scripts to generate memory-access traces. To prepare the inputs for SpMV-based applications, we use Matlab to preprocess our sparse matrices, listed in Table III, obtained from the SuiteSparse collection [27], six from the scientific-computing domain and six graphs.

Our baseline NDP designs for embedding lookup are TensorDIMM [9] and RecNMP [8], and for SpMV-based applications is the Two-Step algorithm [10]. The Two-Step [10] algorithm is the state-of-the-art NDP accelerator for SpMV, which converts random memory accesses to regular accesses and ensures full memory streaming. The Two-Step algorithm mostly focuses on optimizing the implementation of the merge step (i.e., iterations>0 in Figure 8) by using a binary tree-based multi-way merge core. The main contribution of the Two-Step algorithm is parallelizing the multi-way merge operation to handle *large* and *highly sparse* graphs. To reproduce the performance numbers of preceding NDP accelerators, we implement them on our FPGA platform based on the information/con-

Table IV: Latency (cycles @200MHz) of the components in compute units of Fafnir for FPGA implementation.

	Compare	Parallel paths (reduce or forward)			
		Reduce (value)	Reduce (header)		Forward
			indices	queries	
per item (iteration)	12	3	4	3	16
batch size = 8/16/32	N/A		32/64/128	29/53/101	N/A

figurations provided in their published papers. We validate the reproduced numbers against their reported numbers.

VI. Performance Evaluation

This section qualitatively evaluates the performance of Fafnir in terms of latency, end-to-end speedup, scalability, energy-savings, and power/area overhead.

Latency: First, Table IV lists the latency of the compute-unit components that define the latency of pipeline stages and the critical path for our FPGA implementation @200MHz. The critical-path latency is defined by the latency of the compare and reduce units (since reduce and forward are parallel and reduce is slower). Before investigating the key metrics that are end-to-end speedup, scalability, and energy, we quantitatively explore the challenges introduced in Section III by comparing the single-query latency of Fafnir with baselines. To do so, we measure the latency of a query, including random accesses to 16 512B vectors distributed over 32 ranks (4× channels, 4× DIMMs, 2× ranks). Figure 11 shows the contribution of memory access and computation (reduction operation) in total latency. Several parameters such as vector size and number in a query, row buffer size, distribution of vector, the number of pipeline stages, and DRAM timing define the effectiveness of (i) benefiting from row-buffer locality and (ii) not relying on spatial locality, on computation and memory latency. For instance, as Figure 11 illustrates, the computation latency of TensorDIMM, which *pipelines* the processing of 16 embedding vectors in a query, is 2.5× slower than Fafnir, which processes all 16 vectors in *parallel.* Although the parallelism level of RecNMP is also similar to that of Fafnir, its computation latency is not as low as in Fafnir because RecNMP forwards a few (here ∼ 25%) computations to the CPUs as a result of lack of spatial locality. In terms of memory latency, however, Fafnir and RecNMP are identical since they similarly utilize rank-level parallelism and row-buffer hit. In this example, the memory latency of TensorDIMM is 4.45× slower than RecNMP and Fafnir, which could be up to 16× slower in the case of no row buffer hit.

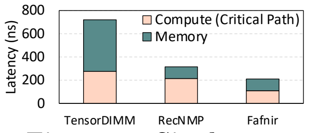


Figure 11: Single-query latency breakdown.

End-to-end Inference Speedup: To evaluate the impact of accelerating the embedding lookup on the overall inference latency, Figure 12 shows the end-to-end speedup of RecNMP and Fafnir over the baseline (1-rank) when increasing ranks from two to 32. The figure shows the breakdown of total inference latency into three components:

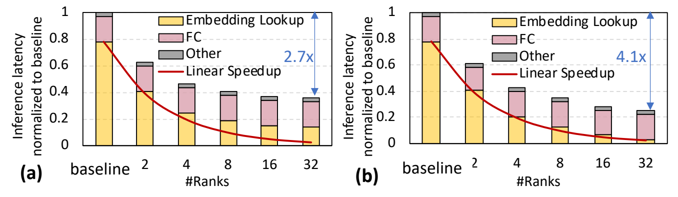


Figure 12: End-to-end inference speedup for DLRM on Kaggle (batch size = 8): (a) RecNMP, (b) Fafnir.

(i) embedding lookup; (ii) fully-connected (FC) layers executed at CPU, the performance of which is assumed to be fixed in various ranks. In Figure 12, FC layers take 0.5 *ms*, however, their latency varies significantly based on the host system (CPU vs. GPU) and batch size [7] – optimizing the performance of FC layers is not the focus of this paper; and (iii) other operations. While both RecNMP and Fafnir work close to the ideal linear speedup (red line) for fewer ranks, Fafnir keeps following the red line more closely as the number of ranks increases to 32. This stems from the key difference between RecNMP and Fafnir: the DIMM-level parallelism by putting a small chip (channel node) between memory and the core to perform *all* reductions at NDP rather than in the cores, the impact of which is more pronounced in larger memory systems with more ranks.

Scalability: In a scalable design, increasing the batch size must help increase throughput. To evaluate the impact of concurrent batch processing on scalability, Figure 13 illustrates the speedup over RecNMP when batch size varies. Although all three designs utilize batch processing, their difference is in the hardware mechanism to most effectively take advantage of a batch to improve throughput. As Figure 13 illustrates, RecNMP looks up embedding approximately 15× faster than TensorDIMM. This speedup stems from the approach of RecNMP to utilize rank-level parallelism. As Figure 13 shows, the speedup of Fafnir over RecNMP, however, more significantly grows with the batch size (i.e., 3.1×, 6.7×, and 12.3×, for batch size 8, 16, and 32, respectively) when neither Fafnir nor RecNMP eliminates redundant memory accesses. The reason is that Fafnir better utilizes memory bandwidth, therefore, filling the gap under the roofline model of RecNMP by performing full-reduction near memory. The tiny (i.e., 0.121 mm^2) channel-node chip between the memory channels and core

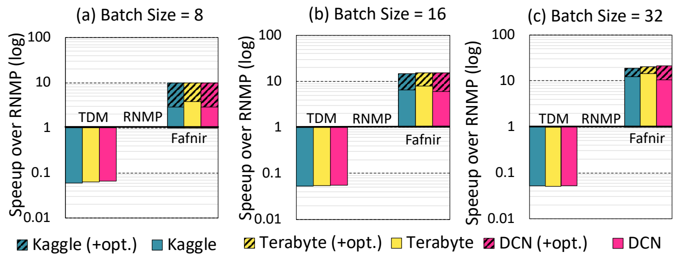


Figure 13: Speedup of Fafnir and TensorDIMM [9] (TDM) over RecNMP [8] (RNMP) for batch sizes (a) 8, (b) 16, and (c) 32. Opt. stands for the optimization of elimination of the redundant memory accesses.

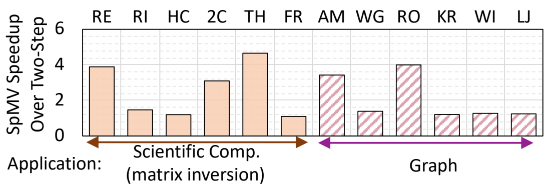


Figure 14: Speedup of Fafnir over Two-Step algorithm [10] for two SpMV-based applications: scientific computations (matrix inversion algorithm) and graph.

is the key to achieve this. In addition, as the striped part of Figure 13 shows, Fafnir achieves up to an extra 3.4× speedup by more effectively eliminating redundant accesses to memory without using caches. For RecNMP, we assume 128KB rank caches that offer the optimal hit rate of 50%.

Other Applications: We also evaluate the speedup of Fafnir over the state of the art for two SpMV-based applications. While Fafnir performs the first step (iteration 0) of SpMV more quickly, the Two-Step algorithm more quickly merges the result (iterations >0). This is because, unlike the Two-Step algorithm, Fafnir does not rely on decompression mechanisms and is able to apply SpMV on data as it is streamed from memory. Further, instead of a chain of adders connected to multipliers, Fafnir uses the tree for the reduction. Conversely, since the Two-Step algorithm particularly optimizes the merge operation, it performs the merge steps more quickly. Because of the mentioned reasons, as Figure 14 illustrates, with no modifications in hardware, Fafnir can process SpMV-based sparse problems more quickly (e.g., up to 4.6×) or in the worst case as quickly as (e.g. 1.1×) the Two-Step. For smaller matrices, as fewer merge iterations are required, Fafnir performs more quickly than larger ones. In some workloads among the larger matrices (e.g., RO) sparseness is a reason that makes them more suitable for Fafnir. Based on our observation, a promising future direction is the combination of both Fafnir (for the first step) and Two-Step (for merging).

Memory Energy Saving: Given that the energy consumption of DRAM dominates that of computation, the energy-savings of memory is essential. Fafnir promises memory energy savings by eliminating extra memory accesses *without using any caching mechanism*. More specifically, Fafnir saves 34%, 43%, and 58% memory accesses for batch

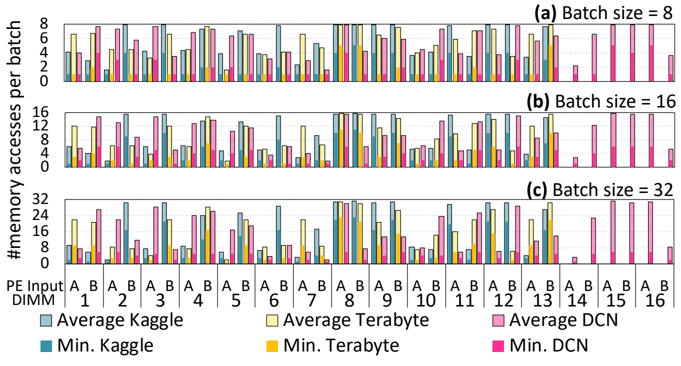


Figure 15: Number of memory accesses at different DIMMs for three batch sizes:(a) 8, (b) 16, and (c) 32.

Table V: FPGA resource utilization for Fafnir.

Resources	DIMM/Rank Node units	DIMM/Rank Node utilization(%)	Channel Node units	Channel Node utilization.(%)
LUT	11800	1.00	7214	0.61
LUTRAM	192	0.03	96	0.02
FF	4646	0.2	3295	0.14
BRAM	68	3.15	26	1.2

sizes 8, 16, and 32, respectively. Figure 15 illustrates the number of memory accesses after eliminating redundant accesses and shows that the number of memory accesses per each input to the leaf PEs is always lower than the batch size (8, 16, and 32 in Figure 15a, 15b, and 15c).

Power Consumption & Area Overhead: This section evaluates the hardware of Fafnir (assuming $n = m = 32$ in Figure 5 and 32 compute units at PEs). Table V lists the resource utilization of Fafnir implementation on FPGA. To embed Fafnir in a standard DIMM-based memory system including four channels, each with four DIMMs, including two ranks, we need four DIMM/rank nodes and one channel node. The implementation of such a system utilizes up to 5%, 0.15%, 1%, and 13% of LUTs, LUTRAMs, FFs, and BRAM blocks of the target FPGA. Figure 16a shows the breakdown of dynamic power consumption of FPGA @200MHz, in total 0.23 W and 0.18 W for DIMM/rank and channel nodes. For our ASIC design, Figure 16b shows the power distribution of a PE. As shown, power consumption has a uniform distribution, which prevents the creation of a hot spot. As well, the breakdown of the power consumption of our ASIC design is listed in Table VI. Our proposed chips add only 23.82 mW per four DIMMs (i.e., 5.9 mW per DIMM) and in total, 111.64 mW to a four-channel memory system, which is negligible compared to the 13 W power consumed by each DDR4 DIMMs, calculated based on a Micron power calculator [9], [31]. As another comparison point, a processing unit RecNMP [8] adds 184.2 mW to one DIMM (estimated at 40 nm @250 MHz).

Table VI lists the area of PE and two types of nodes in Fafnir. A PE is 0.077 mm^2 (including the multiplication units for leaf PE to support SpMV) and the area of DIMM/rank and channel nodes is 0.282 mm^2 (which is smaller than the 0.077×7), and 0.121 mm^2, respectively. Therefore, a benefit of embedding PEs into one chip (as we do) rather than distributing them across DIMMs is a more efficient area. Based on these numbers, we add a total area

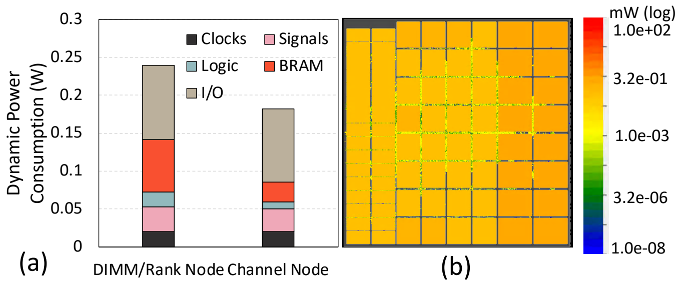


Figure 16: (a) Dynamic power consumption breakdown of Fafnir on FPGA. (b) Power distribution of a PE in our ASIC design at 7 nm.

Table VI: Area and power consumption breakdown @500MHz to switching (Sw.), interconnections (Int.), and leakage (Lkg.) for ASIC design of Fafnir @7 nm.

		PE	DIMM/Rank	Channel
Power(mW)	Sw.	2.1	3.6	2.7
	Int.	8.7	20.1	13.61
	Lkg.	0.02	0.12	0.06
Area(mm^2)		0.077	0.282	0.121

overhead of 1.2 mm^2 to a memory system of 32 ranks. As a comparison point, the area of prior work, a RecNMP [8] processing unit, is estimated as 0.54 mm^2 at 40 nm per one DIMM (8.64 mm^2 to entire 16 DIMMs).

VII. Related Work

Here we explore the prior studies that focus on accelerating graph and HPC problems, classic application domains that contain *sparse gathering*. The majority of the operations in such problems (e.g., 80%) are related to sparse gathering [32]. To relax sparse gathering in graph applications, batching the accesses to the output vector and restricting them to a localized region of memory [33] has been proposed. In addition, several NDP studies [12]–[14], [16], [17] have proposed offloading computation to memory to reduce data movement and leverage NDP to accelerate data access and facilitate computations on sparse data structures [11], [34], [35]. Additionally, DIMMNet [36] has been proposed to accelerate gathering irregular memory accesses. Such NDP solutions, however, are not very effective for embedding lookup of recommendation systems for several reasons. First, they decrease data movement by *rearrangement*, but do not perform *reduction* operations. Second, they are costly, as they copy a page to another in a scratchpad memory (e.g., [32]). Finally, they are not transparent to the software. In addition, several other proposals have accelerated sparse problems by focusing on *computation* [25], [37], [38] for a specific application or technology [39], [40] or using other approaches to reduce the number of accesses to memory accesses [41], [42].

VIII. Conclusions & Future Work

This paper proposed Fafnir, a DDR-based NDP solution for accelerating embedding lookup, the bottleneck-prone task in recommendation systems. The key component of Fafnir is a near-memory intelligent reduction tree, which provides a generic solution for any *sparse gathering*. Besides embedding lookup and SpMV, sparse gathering is also a required function in numeric algebra such as matrix inversion and differential-equation solvers. The particular patterns of computation in such applications necessitate some additional connections in the structure of a tree, which will be envisioned in our future work. The same idea of Fafnir can also be integrated with High Bandwidth Memory (HBM) by connecting the leaf PEs to the 32 pseudo channels rather than the ranks. Such an integration is another direction of our future work.

Acknowledgements

We gratefully acknowledge the support of Booz Allen Hamilton Inc. and the Laboratory for Physical Sciences (LPS), and thank Dr. Eric Cheng from LPS who provided insight and expertise that greatly assisted this research.

References

[1] J. G. Lewis and H. D. Simon, "The impact of hardware gather/scatter on sparse gaussian elimination," *SIAM Journal on Scientific and Statistical Computing*, vol. 9, no. 2, pp. 304–311, 1988.

[2] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM conference on recommender systems*, 2016, pp. 191–198.

[3] M. Naumov, D. Mudigere, H.-J. M. Shi, J. Huang, N. Sundaraman, J. Park, X. Wang, U. Gupta, C.-J. Wu, A. G. Azzolini *et al.*, "Deep learning recommendation model for personalization and recommendation systems," *arXiv preprint arXiv:1906.00091*, 2019.

[4] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, and T.-S. Chua, "Neural collaborative filtering," in *Proceedings of the 26th international conference on world wide web*, 2017, pp. 173–182.

[5] R. Wang, B. Fu, G. Fu, and M. Wang, "Deep & cross network for ad click predictions," in *Proceedings of the ADKDD'17*, 2017, pp. 1–7.

[6] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu, "Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference," *ISCA*, 2020.

[7] U. Gupta, X. Wang, M. Naumov, C.-J. Wu, B. Reagen, D. Brooks, B. Cottel, K. Hazelwood, B. Jia, H.-H. S. Lee *et al.*, "The architectural implications of facebook's dnn-based personalized recommendation," 2020.

[8] L. Ke, U. Gupta, C.-J. Wu, B. Y. Cho, M. Hempstead, B. Reagen, X. Zhang, D. Brooks, V. Chandra, U. Diril *et al.*, "Recnmp: Accelerating personalized recommendation with near-memory processing," *ISCA*, 2020.

[9] Y. Kwon, Y. Lee, and M. Rhu, "Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning," in *MICRO*, 2019, pp. 740–753.

[10] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti, "Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 347–358.

[11] J. C. Beard, "The sparse data reduction engine: chopping sparse data one byte at a time," in *Proceedings of the International Symposium on Memory Systems*, 2017, pp. 34–48.

[12] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C.-Y. Cher, C. H. Costa, J. Doi, C. Evangelinos *et al.*, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 17–1, 2015.

[13] Y. Zhuo, C. Wang, M. Zhang, R. Wang, D. Niu, Y. Wang, and X. Qian, "Graphq: Scalable pim-based graph processing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 712–725.

[14] M. Zhang, Y. Zhuo, C. Wang, M. Gao, Y. Wu, K. Chen, C. Kozyrakis, and X. Qian, "Graphp: Reducing communication for pim-based graph processing with efficient data partition," in *HPCA*. IEEE, 2018, pp. 544–557.

[15] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "Lodestar: Creating locally-dense cnns for efficient inference on systolic arrays," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–2.

[16] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, "Graphpim: Enabling instruction-level pim offloading in graph computing frameworks," in *HPCA*. IEEE, 2017, pp. 457–468.

[17] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 105–117.

[18] B. Asgari, R. Hadidi, N. S. Ghaleshahi, and H. Kim, “Pisces: power-aware implementation of slam by customizing efficient sparse algebra,” in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[19] C. A. Gomez-Uribe and N. Hunt, “The netflix recommender system: Algorithms, business value, and innovation,” *ACM Transactions on Management Information Systems (TMIS)*, vol. 6, no. 4, pp. 1–19, 2015.

[20] H.-J. M. Shi, D. Mudigere, M. Naumov, and J. Yang, “Compositional embeddings using complementary partitions for memory-efficient recommendation systems,” *arXiv preprint arXiv:1909.02107*, 2019.

[21] A. Ginart, M. Naumov, D. Mudigere, J. Yang, and J. Zou, “Mixed dimension embeddings with application to memory-efficient recommendation systems,” *arXiv preprint arXiv:1909.11810*, 2019.

[22] R. Hwang, T. Kim, Y. Kwon, and M. Rhu, “Centaur: A chiplet-based, hybrid sparse-dense accelerator for personalized recommendations,” *ISCA*, 2020.

[23] SciPy, “List-of-list sparse matrix,” https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.lil_matrix.html, 2020, [Online; accessed April-2020].

[24] U. Gupta, B. Reagen, L. Pentecost, M. Donato, T. Tambe, A. M. Rush, G.-Y. Wei, and D. Brooks, “Masr: A modular accelerator for sparse rnns,” in *28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. IEEE, 2019, pp. 1–14.

[25] Z. Zhang, H. Wang, S. Han, and W. J. Dally, “Sparch: Efficient architecture for sparse matrix multiplication,” *arXiv preprint arXiv:2002.08947*, 2020.

[26] B. Asgari, R. Hadidi, and H. Kim, “Ascella: Accelerating sparse computation by enabling stream accesses to memory,” *Proceedings of the 23rd Design, Automation, and Test in Europe (DATE)*, 2020.

[27] T. A. Davis and Y. Hu, “The university of florida sparse matrix collection,” *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, p. 1, 2011.

[28] ASU and ARM, “Arizona State Predictive PDK,” http://asap.asu.edu/asap/, 2020, [Online; accessed April-2020].

[29] “Criteo AI Labs Ad Kaggle,” https://www.kaggle.com/c/criteo-display-ad-challenge, 2020, [Online; accessed April-2020].

[30] “Criteo AI Labs Ad Terabyte,” https://labs.criteo.com/2013/12/download-terabyte-click-logs/, 2020, [Online; accessed April-2020].

[31] Micron, “Calculating memory power for ddr4 sdram,” 2017.

[32] A. Rodrigues, M. Gokhale, and G. Voskuilen, “Towards a scatter-gather architecture: hardware and software issues,” in *Proceedings of the International Symposium on Memory Systems*, 2019, pp. 261–271.

[33] M. Kumar, M. Serrano, J. Moreira, P. Pattnaik, W. P. Horn, J. Jann, and G. Tanase, “Efficient implementation of scatter-gather operations for large scale graph analytics,” in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016, pp. 1–7.

[34] M. Gokhale, S. Lloyd, and C. Hajas, “Near memory data structure rearrangement,” in *Proceedings of the 2015 International Symposium on Memory Systems*, 2015, pp. 283–290.

[35] S. Lloyd and M. Gokhale, “Near memory key/value lookup acceleration,” in *Proceedings of the International Symposium on Memory Systems*, 2017, pp. 26–33.

[36] N. Tanabe, B. Nuttapon, H. Nakajo, Y. Ogawa, J. Kogou, M. Takata, and K. Joe, “A memory accelerator with gather functions for bandwidth-bound irregular applications,” in *Proceedings of the 1st Workshop on Irregular Applications: Architectures and Algorithms*, 2011, pp. 35–42.

[37] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, “Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training,” in *HPCA*, 2020.

[38] B. Asgari, R. Hadidi, T. Krishna, H. Kim, and S. Yalamanchili, “Alrescha: A lightweight reconfigurable sparse-computation accelerator,” in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 249–260.

[39] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, “Scnn: An accelerator for compressed-sparse convolutional neural networks,” *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.

[40] L. Song, Y. Zhuo, X. Qian, H. Li, and Y. Chen, “Graphr: Accelerating graph processing using reram,” in *HPCA*. IEEE, 2018, pp. 531–543.

[41] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, “Extensor: An accelerator for sparse tensor algebra,” in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 319–333.

[42] D. Fujiki, N. Chatterjee, D. Lee, and M. O’Connor, “Near-memory data transformation for efficient sparse matrix multi-vector multiplication,” in *SC*, 2019, pp. 1–17.