A High Throughput Parallel Hash Table Accelerator on HBM-enabled FPGAs

Yang Yang, Sanmukh R. Kuppannagari and Viktor K. Prasanna
Department of Electrical and Computer Engineering
University of Southern California
Email: {yyang172, kuppanna, prasanna}@usc.edu

***Abstract*—Hash table is a key component in a number of AI algorithms such as Graph Convolutional Neural Networks, Approximate Nearest Neighbor Search, Bag-of-Words based Text Mining algorithms, etc. Efficient implementation of hash tables is needed for a wide range of AI applications. High bandwidth memory (HBM), which provides significantly higher memory bandwidth than traditional DDR, has recently gained popularity. In this work, we propose a high throughput parallel hash table targeting HBM-enabled FPGAs. Our design is tailored for HBM architecture, allowing flexible and balanced mapping between processing engines and HBM channels at design time given query distribution and hash table properties (key/value length, collision handling, and hash table size). We further develop a novel data organization and query flow which enable our accelerator to scale up to 16 processing engines (PEs). The proposed design supports parallel search, insert, and delete queries. Experimental results demonstrate that our hash table accelerator can achieve up to 3575 million operations per second (MOPS) for search-only queries and up to 1470 MOPS for 50%/50% distributed search/update queries on HBM-enabled FPGAs. It achieves better throughput than the state-of-the-art GPU and FPGA designs by up to $3.5\times$ and $3.2\times$ respectively.**

I. Introduction

Hash table is a data structure which enables efficient data access and storage for sparse data. It has been adopted to accelerate several emerging AI applications such as graph machine learning, deep learning, data mining, etc. [1]–[5]. For instance, in [5], Chen et al. show that locality sensitive hashing can be used to reduce the amount of neural network computations by selecting a subset of neurons to be activated based on input samples. Graph Convolution Neural Network uses hash tables in the graph sampling operation to determine whether the currently sampled vertex or edge exists in the sampled set [4].

FPGAs have gained a lot of attention due to their immense flexibility and high energy efficiency [6]. The vast compute capability offered by the state-of-the-art FPGAs require large memory bandwidth to supply data to the processing elements. High-bandwidth memory (HBM) has been deployed by leading FPGA vendors to overcome this challenge [7], [8]. Latest HBM-enabled FPGAs can provide up to 512 GB/s bandwidth and 16 GB capacity, which makes them an attractive choice for accelerating data intensive applications.

However, it is non-trivial to efficiently utilize the high bandwidth offered by HBM. Certain unique features of HBM architecture such as the presence of many pseudo-channels and access to limited memory space by each pseudo-channel (Section III-B) can have undesirable performance implications. Naive approach of partitioning hash table across all the channels may require a complex and expensive crossbar between the processing logic and the HBM channels, which is not a scalable solution. On the other hand, creating as many hash table copies as the HBM channels could lead to a simpler design, but it may adversely hurt performance due to the inter-PE communication overheads.

In this paper, we design a parallel hash table accelerator tailored specifically for HBM-enabled FPGAs. It can process p concurrent queries ($p \geq 1$) per cycle. The proposed accelerator assigns dedicated processing engine (PE) and hash table copy to one or more adjacent HBM channels. This avoids complex memory interface between the PEs and the HBM. The flexible architecture enables balanced designs given query distribution, key/value length, collision handling, and hash table size. We further develop novel data organization and query flow technique targeting HBM that can scale our accelerator up to 16 PEs. Our design implements a dynamic hash table that allows any mixture of search, insert and delete queries. It supports parallel collision handling, thus query time complexity of $O(1)$ is achieved. The hash table is stored in HBM, allowing it to scale to millions of entries. Our hash table supports the semantics of relaxed consistency to improve performance [9]. The main contributions of this work are:

- We design and implement a parallel hash table accelerator tailored for HBM-enabled FPGAs.
- To reduce the inter-PE communication overheads and improve throughput, the proposed accelerator allows flexible mapping between processing engines and HBM channels at design time.
- We develop novel data organization and query flow technique specifically targeting HBM architecture to improve throughput and scalability.
- Our hash table supports all common hash table queries, search, insert, and delete and can scale up to 16 pipelines.
- We perform detailed experiments using a wide range of key/value length, hash table size, and query distribution on an HBM-enabled FPGA. Results show that our hash table can sustain a throughput of up to 3575 MOPS for search-only queries and up to 1470 MOPS for 50%/50%

distributed search/update[1] queries.

II. Related Work

Istvan et al. [10] proposed a pipelined hash table architecture for Memcached applications which can sustain 10 Gbps throughput. Cho et al. [11] used bloom filter to reduce unnecessary memory accesses. Tong et al. [12] developed a hash table targeting networking applications that achieves up to 85 Gbps throughput. In [13], the authors developed an architecture that stores keys on-chip and values on DRAM thereby requiring DRAM access when there is a match. All the above works focus on improving the performance of a single processing pipeline. The high bandwidth of HBM cannot be saturated by a single processing pipeline, hence, these techniques cannot be trivially ported to HBM. Pontarelli et al. [14] presented an FPGA-based Cuckoo hash table with multiple parallel pipelines with each pipeline responsible for a single cuckoo function. The scalability of the design is limited by the number of cuckoo functions which is typically very small. Moreover, queries using the same function at the same time are serialized, thus significantly reducing the throughput.

In [9] the authors developed a parallel hash table using on-chip SRAMs of FPGA. For p pipelines, they make p copies of the hash table to maximize throughput. However, the design can support only small hash tables ($<$ 200 K entries in most recent FPGAs) and only search and insert queries. Moreover, their replication is query distribution agnostic and when ported to HBM leads to reduced throughput (Section IV). Our work addresses these limitations as follows: (i) Using HBM for hash table dramatically increases the size of the tables that can be supported, (ii) we support all hash table query types — search, insert, and delete, (iii) we perform detailed experimental analysis to show that naively increasing hash table copies may adversely hurt throughput.

III. Hash Table on FPGA

A. Hash Table Semantics and Supported Queries

A hash table uses one or more hash functions to map keys into an array of entries, from which the desired value can be read and/or updated [15]. Our design implements the standard closed addressing hash table. For collision resolution, we implement the separate chaining method. Every hash table entry has multiple slots, each of which can store one key-value pair. The number of slots is pre-determined at design time. If number of key-value pairs that are mapped to one entry exceeds the number of slots, our design raises an abort error. Our hash table accelerator supports the same relaxed consistency proposed by [9]. The key observation is that by not using complicated forwarding units for address conflict handling, inconsistency in hash table may occur. However, the maximum number of such erroneous queries is bounded. More details on the consistency model can be found in [9]. The hash table queries supported by our accelerator and their semantics are defined below.

[1]In this paper, we use hash table *update* to include both insert and delete queries.

- SEARCH (k): Return $\{k, v\} \in S$ or $\varnothing$. Retrieve the value associated with the input key if the key exists in the hash table, or empty if not found.
- INSERT (k, v): $S \leftarrow (S - \{k, *\}) \cup \{k, v\}$. Insert a new key-value pair to the hash table if the key does not exist or replace the previously inserted value for this key with a new value.
- DELETE (k, v): $S \leftarrow S - \{k, v\}$. Delete the key and value from the hash table if they exist.

Parallel Hash Table is an implementation of hash tables that allows concurrent accesses. Instead of operating on one key at a time, each query can take a set of keys and perform hash table operations at the same time.

B. Target Platform - FPGA + HBM2

High-bandwidth memory (HBM) vertically stacks multiple DRAM dies and connects them using through-silicon vias (TSVs) [16]. Each stack of HBM2 (2nd generation of HBM) can have up to 8 DRAM dies (channels). Each memory channel has a memory controller (MC), and is further divided into two pseudo-channels (PCs). Each PC has a dedicated HBM physical interface, which is used to send/receive memory request/response. The peak bandwidth that can be achieved from an HBM2 stack is 256 GB/s [8]. On the application side, a configurable number of ports (typically up to 16 for each HBM stack) can be enabled to access HBM memory simultaneously. A switch fabric is required to route memory access from the application to each PC. Leading FPGA vendors offer a switch fabric as a drop-in solution to ease the design complexity. For instance, Xilinx has opted to provide a hard switch fabric to enable access to the full HBM address space from any port on the application side [7]. However, the application bandwidth to access non-local PCs is limited by the lateral connections of the switch fabric. On the other hand, Intel FPGAs provide a soft AXI switch fabric to access HBM. The mapping between the application ports and the PC ports is one-to-one [8].

C. Accelerator Design

Our goal is to develop a high-throughput hash table with p Processing Engines (PEs) on HBM-enabled FPGAs. Each PE can process all the supported queries concurrently. The physical architecture of HBM and the limitations of switch fabric aforementioned makes it non-trivial to port existing techniques such as partitioning [14] or replication [9]. To address these challenges, our accelerator allows a flexible connection from each PE to one or more adjacent HBM channels. This architecture choice avoids complex and expensive PE-HBM memory interface. It can also reduce inter-PE communication overheads at design time given query distribution, key/value length, and hash table size.

1) Data Organization and Query Flow: To fully utilize the HBM bandwidth, we divide the HBM PCs into multiple groups, and each PE is connected to a group of PCs. The number of PCs that is assigned to each PE is an architecture parameter that can be decided at design time. The multi-PE

design allows the accelerator to improve HBM bandwidth utilization. We further store a copy of the entire hash table to each group of PCs. Hash table entries are evenly distributed among the PCs within each group. To enable hash table updates that are performed by a PE are properly propagated globally, a ring interconnect is used for inter-PE communication. Figure 1 shows an example of our proposed hash table data organization with 4 PEs (hash table copies). Each PE is connected to 2 HBM channels (4 PCs), where one hash table copy is stored.

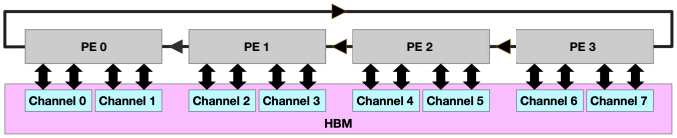
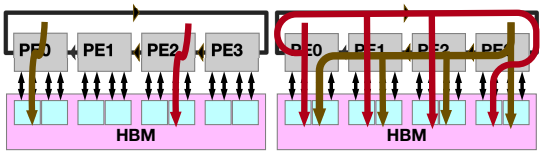


Fig. 1. Hash Table Accelerator Architecture with 4 PEs.

Our hash table accelerator supports search, insert, and delete queries defined in Section III-A. The query flow of our hash table, i.e., mapping of search, insert, and delete operations to our parallel architecture, is described below:



(a) (b)

Fig. 2. Query dataflow. Colored arrows show how queries flow through the proposed accelerator. (a) Search query flow: PE0 and PE2 process 2 concurrent queries. (b) Insert and delete query flow: PE0 and PE3 process 2 concurrent queries.

Search: A search query, received by a PE, goes through its *Request Pipeline* which issues HBM read to fetch the hash table entry. Note that one entry can have multiple hash table slots. The input key is checked against the keys in all the slots (if they are valid), and if there is a match, the value is returned to the application through *Response Pipeline*. *Request Pipeline* and *Response Pipeline* are described in the next subsection. As shown in Figure 2(a), search queries are handled within a PE completely.

Insert/Delete: An insert/delete query first checks the existence of the key and then modifies it. Thus this query starts with an HBM read operation. After that, p HBM writes to the local and remote hash table copies are required, where p equals to the number of PEs. The write operation updates the key-value pair in the hash table according to the query type and the matching result. A ring interconnect is used to propagate updates from local PE to remote PE. Figure 2(b) depicts the data flow of insert/delete queries initiated by PE0 and PE3.

2) Processing Engine (PE) Design: Our hash table accelerator consists of an array of homogeneous PEs. Each PE can receive input queries independently. Figure 3 shows the microarchitecture design of each processing engine. Inside a PE, two pipelines work concurrently. *Request Pipeline* processes incoming queries and issues HBM read requests; *Response Pipeline* handles the HBM read responses and performs hash table updates (HBM writes). Between the two pipelines, our accelerator uses Pending Request Queue and Response Queue to increase PE's HBM read latency tolerance.

Request pipeline is a 3-stage pipeline (Figure 3). The first stage reads an incoming query from the Input Command Queue, and calculates hash table index for the incoming key. We use the Class H_3 [17] hashing, which has been demonstrated to be effective on distributing keys randomly among hash table entries. The second stage receives the hash table index, along with query command and value to be inserted (for insert queries), and performs resource allocation — checking availability in Pending Request and Response Queues. Pending Request Queue is a per pseudo-channel structure that stores outstanding HBM reads. In the last stage of the *Request Pipeline*, an HBM read request with size equals to one hash table entry is issued to the proper HBM pseudo-channel based on the calculated hash table index. We issue to read the entire entry instead of one slot at a time to improve HBM efficiency and collision handling overhead.

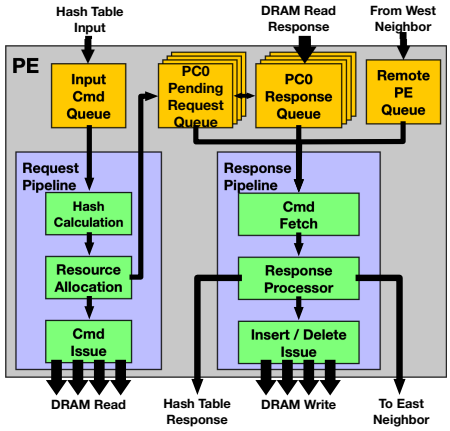


Fig. 3. Processing engine microarchitecture.

Each pseudo-channel is connected to a separate PC Response Queue to store HBM data responses. When the response for an HBM read request arrives, the response data is written to the corresponding PC Response Queue. In the meantime, it also sets the valid bit such that the transaction becomes eligible to be processed by the *Response Pipeline*. We use AXI protocol to enforce in-order response, therefore the ordering between Pending Request Queue and Response Queue for the same pseudo-channel is always in sync. This simplifies the arbitration logic because we can use FIFO instead of reorder buffers for the Pending Request Queues and Response Queues. The Remote PE Queue is used to store the hash table insert/delete queries from remote PEs. Each entry in the Remote PE Queue has information on the destination PE ID, hash table index, slot ID for the key-value pair, and also the key-value pair to be updated.

Response Pipeline is also a 3-stage pipeline. The first stage performs round-robin arbitration between each Pending Request Queue and Remote PE Queue. Once it has a winner, it signals the corresponding queue to pop the entry, and stores the entry in a pipeline register. The second stage, Response

Processor, handles hashing collisions and initiates inter-PE communication. The core component in this stage is a parallel result resolution and collision handling unit, as illustrated in Figure 4. This component determines the slot position (slot id) to be operated on for a given query. Each hash table entry has a fixed number of slots that are stored in continuous memory locations. This enables the Response Processor to be able to check key-value pairs in multiple slots in parallel. For search, this component detects and generates a matching flag to indicate that the query key exists in the hash table. It also populates the value register with the data from HBM. For insert/delete, it further initiates inter-PE communication by creating an insert/delete packet and presenting the packet to the ring interconnect. Collision handling is performed by finding the first available slot of a given entry. Lastly, the third stage of *Response Pipeline* issues HBM write command to update the local hash table content.

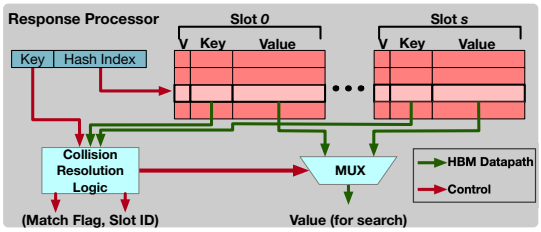


Fig. 4. Parallel result resolution and collision handling.

3) Inter-PE Communication: We design an uni-directional 1D ring interconnect (Figure 1) for inter-PE communication. We use a ring interconnect as it can easily scale to multiple PEs without complex routing logic. Each PE is a stop in the ring interconnect. A packet originates at a source PE and contains information on destination PE ID, hash table index, query type, key, value, and slot ID. It enters the ring interconnect from the second stage of the *Response Pipeline* of the source PE. The packet is then routed to neighbor PE along the direction of the interconnect and stored in Remote PE Queue at each stop. Once it reaches the destination PE, the forwarding stops. Since the arbitration in the *Response Pipeline* is round-robin, it guarantees forward progress of each Remote PE Queue that connects to the ring interconnect. It is worth noting that more PEs enable more parallelism, but they also incur significant HBM bandwidth overheads due to inter-PE communication from insert and delete queries.

IV. Experimental Results

A. Experimental Methodology

We evaluate our hash table accelerator using Intel Stratix 10 MX 2100 FPGA [8]. The FPGA is equipped with two 8GB HBM2 stacks. Each HBM stack has 16 pseudo-channels and can provide up to 256 GB/s bandwidth. Intel Quartus Prime 19.3 is used for synthesis, place-and-route, and system integration. We conduct detailed analysis on the scalability, performance, and resource utilization of the proposed accelerator. We run experiments with different number of PEs (p), key/value length (k/v), slots per entry (s), search to total query ratio (r), memory efficiency (*eff*), as well as hash table size. Key and value of each query are generated randomly. Traffic is injected into each PE independently as long as there is no back-pressure from the memory. We perform post place-and-route simulations to measure the execution time for processing all the queries and use it to calculate the throughput.

To evaluate the performance and scalability of our accelerator under different system load conditions, we employ a fixed latency/bandwidth memory model. The memory model is configured to have 32 ports, which matches the total number of PCs with 2 HBM stacks. Since the minimum frequency of user logic clock is one quarter of the HBM2 interface frequency (f_{hbm}) in Stratix 10 MX 2100 [18], we adjust the memory model frequency based on accelerator's post place-and-route max frequency. Hence, peak bandwidth of each memory port is set to 2 (DDR) $\times$ 64 bits (data width) $\times$ f_{hbm}. An *eff* knob is created in the memory model to reflect the efficiency factor of HBM memory. [19], [20] report less than 200 ns HBM access latency. Although it is based on Xilinx FPGA, the underlying memory technology is the same. However, in practice HBM latency is not fixed, we design our accelerator to have enough buffering to accommodate potential variable latency. Simply assuming 200 ns latency may lead to overly optimistic design, therefore we use 300 ns HBM latency in our experiments (unless otherwise specified) to allow some margin. We use million operations per second (MOPS) as the throughput metric.

B. Results

1) Throughput Evaluation: We evaluate the achieved throughput by varying p, s, r, and *eff*. We conduct two sets of experiments: (i) We fix s and explore the effect of *eff* from 40% to 70%. This range is selected based on existing studies in [20], [21] and, (ii) we fix *eff* and explore the effect of s. The hash table size is fixed at 1 M and the key/value length (k/v) are both 4 bytes. In both sets of experiments, we also sweep r from 30% to 100%. In total, 8 configurations are implemented. The design runs from 221.4 MHz (p=16, s=8) to 231.8 MHz (p=2, s=4) with an average of 225.2 MHz.

Figure 5 shows the results. From each sub-figure, we observe that HBM can be saturated with 8 or more PEs. When this happens, read heavy (larger r) query distribution benefits from large number of PEs. However, as query distribution becomes write heavy, lesser PEs could perform better, especially when the memory efficiency is low. Inter-PE communication overheads are the main reason causing the throughput drop with more PEs. For instance, with 16 PEs running in parallel, the design can easily oversubscribe the HBM. When r=1.0, all the HBM traffic is generated from the *Request Pipeline*. However, when r=0.7, 30% of the queries from each PE need to be sent to all the other PEs for inter-PE communication, which consume a lot of HBM bandwidth. As HBM is already saturated, the inter-PE communication becomes an overhead that reduces the throughput significantly. In contrast, from r=0.7 to r=0.5/0.3, the inter-PE communication overhead does not change as sharply as the previous case (r form 1.0 to 0.7),

therefore the throughput drop is less severe. When using 8 PEs, the available bandwidth per PE is doubled compared to 16 PEs design (4 PCs vs 2 PCs), which provides more margin for inter-PE communication overheads. When comparing sub-figures in the same row, we observe that increasing s reduces throughput whenever the accelerator is limited by HBM bandwidth. This is due to the fact that bytes to be processed for each query are proportional to the number of slots per entry.

In summary, our designs can run at high frequency with up to 16 PEs, proving the scalability of our accelerator. Figure 5 shows clearly that increasing hash table copies as many as possible may adversely hurt throughput due to the inter-PE communication overheads. We obtained balanced designs by considering query distribution, key/value length, slots per entry and hash table size.

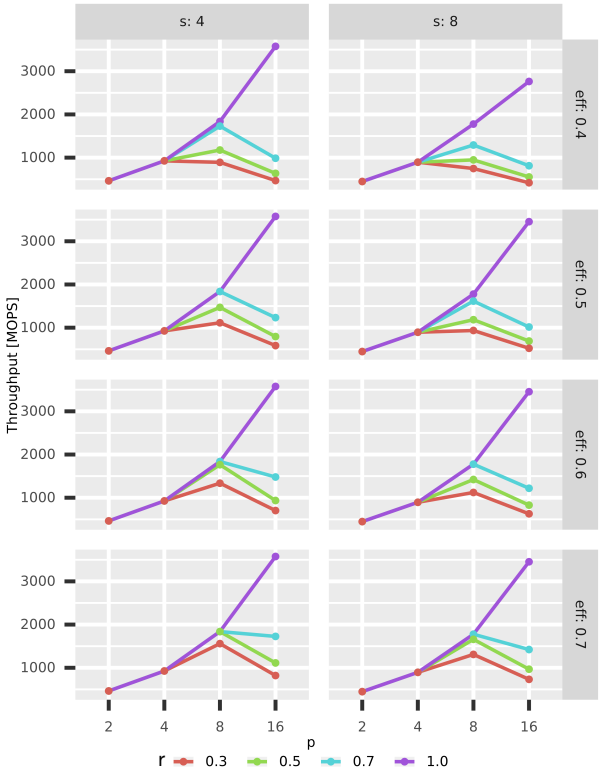


Fig. 5. Achieved throughput with 1 M hash table size, *k/v* = 4B/4B. Varying p, s, r, and *eff*.

2) Latency Evaluation: Depending on the access pattern, DRAM access may take different number of clock cycles. We run experiments to evaluate the latency impact on our design with various *k/v* length. We fix the hash table size to 1 M, configure the accelerator to have 8 PEs, 4 slots per entry, and search ratio of 50%. The *eff* factor is set to 60%. We configure the memory latency to be 300 ns, 450 ns, and 600 ns, which is higher than the latency reported in [19], [20]. As shown in Figure 6(a), our design can tolerate the latency increase in most of the cases. There is a slight performance drop in 4B/4B case. For cases with *k/v* greater than 4B/4B, the throughput is not impacted. This is because in those cases each transaction reads more bytes, and can saturate memory bandwidth with lesser number of outstanding transactions. If the expected latency is too high, one may consider further increase the size of Pending Request Queue and Response Queue, as they are the structures that store outstanding requests. It is set to 128 deep in the proposed accelerator according to calculations based on Little's Law [22].

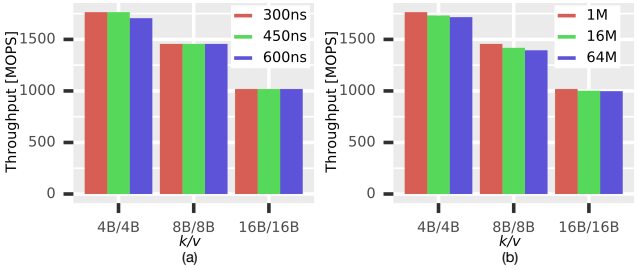


Fig. 6. (a) Achieved throughput with 1 M hash table size, 8 PEs, 4 slots per entry. Varying *k/v* and memory latency. (b) Achieved throughput with 8 PEs, 4 slots per entry. Varying *k/v* and hash table size.

3) Hash Table Size Evaluation: Figure 6(b) shows the performance of our accelerator for various number of hash table sizes. We configure the accelerator to have 8 PEs, 4 slots per entry, and search ratio of 50%. The *eff* factor is set to 60%. We run experiments using 1 M, 16 M, and 64 M hash table size and different *k/v* length. Increasing the hash table size doesn't affect the performance significantly. In our design, only hash index bit-width changes with hash table size. The increase in bit-width follows logarithmic growth which only slightly increases the complexity of hashing unit. As a result, our design demonstrates good scalability with hash table size. Further, since our accelerator implements closed addressing, separate chaining hash table, load factor does not influence the throughput of our hash table.

4) Resource Utilization: Since the hash table is completely stored in HBM, our design does not require a lot of FPGA resources. For 16 PEs, the resource consumption is: ALM $<$ 8%, M20K memory $<$ 1% and no DSP usage. Slots per entry (s), *k/v* length, and hash table size have little impact on FPGA resource consumption.

TABLE I
THROUGHPUT (MOPS) COMPARISON WITH GPU DESIGN

Search Ratio (r)	[23]	Our Design
100% search, 0% update	937	1837
80% search, 20% update	approx. 525	1837
60% search, 40% update	approx. 490	1750
0% search, 100% update	approx. 420	816

C. Comparison with Prior Work

We compare our design with the state-of-the-art implementations on GPU [23] and FPGA [9]. Our hash table is configured to use 1 M hash table size, 4 slots per entry, 4B/4B *k/v* length. This is the closest configuration we find to make a fair comparison. We use 8 PEs design to compare against their work, unless otherwise specified. To be conservative, we choose 50% *eff* factor (256 GB/s HBM bandwidth), which is the random pattern efficiency reported in [21].

1) Comparison with GPU Design: In [23], the design is implemented on NVIDIA Tesla K40c GPU — 2880 CUDA cores operating at 745 MHz (boosted up to 876 MHz) with 288

TABLE II
THROUGHPUT (MOPS) COMPARISON WITH FPGA DESIGN

Hash Table Size	[9]	Our Design (r=0.5)	Speedup
50 K	1628	1470	0.9x
100 K	840	1470	1.7x
200 K	447	1470	3.2x
1 M	n/a	1470	n/a

GB/s off-chip memory bandwidth. Their insert operation does not ensure uniqueness among the keys. Customized APIs have to be used to ensure consistency. Table I shows the comparison results. We compare with the peak achieved throughput on GPU. It is important to note that, unlike our accelerator, their throughput varies widely based on the load factor. Our design achieves better throughput than the GPU implementation with speedup of $1.9\times$, $3.4\times$, $3.5\times$, $1.9\times$ for distributions with 100% search, 80% search, 60% search, 0% search respectively.

2) Comparison with FPGA Design: Yang et al have recently proposed a parallel hash table using FPGA on-chip SRAM [9] which achieves high throughput by using as many hash table copies as possible. Here we perform an empirical comparison. Table II shows the results using the same FPGA in [9]. When hash table size is small, [9] performs marginally better, because of the higher on-chip SRAM bandwidth. As hash table size increases, their design achieves less throughput or even becomes infeasible due to the limitation of SRAM capacity. In contrast, the proposed accelerator uses flexible PE-HBM mapping to reduce inter-PE communication overheads while can support large hash table with millions of entries.

V. CONCLUSION

In this work, we proposed a high throughput parallel hash table accelerator on HBM-enabled FPGAs. It can be used to target a variety of applications in graph analytics, machine learning, and data mining. Our accelerator allows flexible mapping between processing engines (PEs) and HBM channels to balance the throughput at design time. We further developed novel data organization and query flow technique to scale our accelerator to 16 PEs. We performed detailed experimental analysis on the performance of the proposed design by varying number of PEs, key/value length, slots per entry, hash table size, and query distribution. Our evaluation shows that our accelerator can achieve up to 3575 MOPS for search-only queries and up to 1470 MOPS for 50%/50% distributed search/update queries on an HBM-enabled FPGA. Compared with the state-of-the-art designs on GPU and FPGA, the proposed accelerator achieves up to $3.5\times$ and $3.2\times$ speedup in throughput respectively.

ACKNOWLEDGMENT

We would like to thank anonymous reviewers for their valuable feedback. This work is partially supported by Intel Strategic Research Alliance and by the National Science Foundation under grant OAC-1911229.

REFERENCES

[1] C. Boulis and M. Ostendorf, "Text classification by augmenting the bag-of-words representation with redundancy-compensated bigrams," in *Proc. of the International Workshop in Feature Selection in Data Mining*. Citeseer, 2005, pp. 9–16.

[2] S. Vijayanarasimhan and J. Yagnik, "Large-scale classification in neural networks using hashing," Aug. 14 2018, uS Patent 10,049,305.

[3] J. D. Holt and S. M. Chung, "Mining association rules in text databases using multipass with inverted hashing and pruning," in *14th IEEE International Conference on Tools with Artificial Intelligence, 2002.(ICTAI 2002). Proceedings.* IEEE, 2002, pp. 49–56.

[4] H. Zeng, H. Zhou, A. Srivastava, R. Kannan, and V. Prasanna, "Accurate, efficient and scalable graph embedding," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 462–471.

[5] B. Chen, T. Medini, and A. Shrivastava, "Slide: In defense of smart algorithms over hardware acceleration for large-scale deep learning systems," in *Proceedings of the 3rd MLSys Conference (MLSys)*, 2020.

[6] S. R. Kuppannagari, R. Chen, A. Sanny, S. G. Singapura, G. P. C. Tran, S. Zhou, Y. Hu, S. P. Crago, and V. K. Prasanna, "Energy performance of fpgas on perfect suite kernels," in *2014 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2014, pp. 1–6.

[7] Xilinx, "Xilinx UltraScale+ HBM FPGAs," https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus-hbm.html.

[8] Intel, "Stratix 10 MX FPGAs," https://www.intel.com/content/www/us/en/products/programmable/sip/stratix-10-mx.html.

[9] Y. Yang, S. Kuppannagari, A. Srivastava, R. Kannan, and V. Prasanna, "Fasthash: Fpga-based high throughput parallel hash table," in *Proceedings of the 35th International Supercomputing Conference (ISC-HPC)*, 2020.

[10] Z. István, G. Alonso, M. Blott, and K. Vissers, "A flexible hash table design for 10gbps key-value stores on fpgas," in *2013 23rd International Conference on Field programmable Logic and Applications*, 2013, pp. 1–8.

[11] J. M. Cho and K. Choi, "An fpga implementation of high-throughput key-value store using bloom filter," in *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test*, 2014, pp. 1–4.

[12] D. Tong, S. Zhou, and V. K. Prasanna, "High-throughput online hash table on fpga," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015, pp. 105–112.

[13] Wei Liang, Wenbo Yin, Ping Kang, and Lingli Wang, "Memory efficient and high performance key-value store on fpga using cuckoo hashing," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, 2016, pp. 1–4.

[14] S. Pontarelli, P. Reviriego, and J. A. Maestro, "Parallel d-pipeline: A cuckoo hashing implementation for increased throughput," *IEEE Transactions on Computers*, vol. 65, no. 1, pp. 326–331, 2016.

[15] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[16] H. Jun, J. Cho, K. Lee, H. Son, K. Kim, H. Jin, and K. Kim, "Hbm (high bandwidth memory) dram technology and architecture," in *2017 IEEE International Memory Workshop (IMW)*, 2017, pp. 1–4.

[17] J. L. Carter and M. N. Wegman, "Universal classes of hash functions (extended abstract)," in *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '77. New York, NY, USA: ACM, 1977, pp. 106–112.

[18] Intel, "High bandwidth memory (hbm2) interface intel fpga ip user guide," https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ug/ug-20031.pdf.

[19] Xilinx, "Xilinx axi high bandwidth memory controller v1.0," 2019.

[20] Z. Wang, H. Huang, J. Zhang, and G. Alonso, "Shuhai: Benchmarking high bandwidth memory on fpgas," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2020, pp. 111–119.

[21] D. Kehlet, O. Tan, and L. Landis, "Intel stratix 10 mx fpgas revolutionizing system memory bandwidth," *FCCM Workshop on Inte FPGA New Technology Showcase: Chiplets, High-Bandwidth memory, and eASICS*, 2019.

[22] J. L. Gustafson, *Little's Law*. Boston, MA: Springer US, 2011, pp. 1038–1041.

[23] S. Ashkiani, M. Farach-Colton, and J. D. Owens, "A dynamic hash table for the gpu," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, May 2018, pp. 419–429.