FASTHash: FPGA-Based High Throughput Parallel Hash Table

Yang Yang[1(✉)], Sanmukh R. Kuppannagari[2], Ajitesh Srivastava[2], Rajgopal Kannan[3], and Viktor K. Prasanna[2]

[1] Department of Computer Science, University of Southern California, Los Angeles, CA, USA
yyang172@usc.edu
[2] Ming Hsieh Department of Electrical and Computer Engineering, University of Southern California, Los Angeles, CA 90089, USA
{kuppanna,ajiteshs,prasanna}@usc.edu
[3] US Army Research Lab, Playa Vista, Adelphi, CA 90094, USA
rajgopal.kannan.civ@mail.mil

Abstract. Hash table is a fundamental data structure that provides efficient data store and access. It is a key component in AI applications which rely on building a model of the environment using observations and performing lookups on the model for newer observations. In this work, we develop FASTHash, a "truly" high throughput parallel hash table implementation using FPGA on-chip SRAM. Contrary to state-of-the-art hash table implementations on CPU, GPU, and FPGA, the parallelism in our design is data independent, allowing us to support p parallel queries ($p > 1$) per clock cycle via p processing engines (PEs) in the worst case. Our novel data organization and query flow techniques allow full utilization of abundant low latency on-chip SRAM and enable conflict free concurrent insertions. Our hash table ensures relaxed eventual consistency - inserts from a PE are visible to all PEs with some latency. We provide theoretical worst case bound on the number of erroneous queries (true negative search, duplicate inserts) due to relaxed eventual consistency. We customize our design to implement both static and dynamic hash tables on state-of-the-art FPGA devices. Our implementations are scalable to 16 PEs and support throughput as high as 5360 million operations per second with PEs running at 335 MHz for static hashing and 4480 million operations per second with PEs running at 280 MHz for dynamic hashing. They outperform state-of-the-art implementations by 5.7x and 8.7x respectively.

Keywords: Hash table · Parallel processing · FPGA

1 Introduction

Artificial Intelligence (AI) has played a central role in pushing the frontiers of technology. There has been a significant progress in several domains due to AI,

© Springer Nature Switzerland AG 2020
P. Sadayappan et al. (Eds.): ISC High Performance 2020, LNCS 12151, pp. 3–22, 2020.
https://doi.org/10.1007/978-3-030-50743-5_1

including computer vision [3], robotics [21], machine learning on graphs [32], and games [8]. Conceptually, AI algorithms use observations to *learn* a model for the task, which is then consulted (searched, looked-up) to reason about new observations and update the model. To enable fast look-up during training as well as inference, hash table has been widely adopted in the implementation of many AI algorithms [10,13,27,28,32]. For example, Graph Convolution Neural Network (GCN) uses hash tables in the graph sampling operation to determine whether the currently sampled vertex or edge exists in the sampled set or not [32]. Similarly, in Approximate Nearest Neighbor (ANN), hashing is used to determine the neighbor (point) closest to the current observation [28]. Hashing plays a central role in text-mining for creating and maintaining bag-of-words models [4]. Therefore, a parallel high throughput hash table is imperative to accelerate a wide range of AI applications.

Several works have developed high-throughput hash table implementations by "parallelizing" the hash table. The "parallelization" in these works implies exploiting certain features of the hash table, such as the availability of multiple partitions [22], to increase the number of parallel queries that can be supported. However, this does not imply *true parallelism* as the parallelism is highly data dependent and the worst case performance - for example, when all queries belong to the same partition - is similar to a serial implementation. In contrast, our focus in this work is to develop a parallel implementation of hash table that supports p parallel queries ($p > 1$) in each clock cycle even in the worst case.

Field Programmable Gate Arrays (FPGA) have proved successful in applications which require energy-efficient acceleration of complex workloads such as AI due to their high energy-efficiency and the availability of fine grained parallelism [18]. Their dense logic elements (up to 5.5 million), abundant user-controllable on-chip SRAM resources (up to 500 MB, up to 38 TB/s bandwidth), and interfaces with various external memory technologies such as High Bandwidth Memory (HBM) make them a logical choice for accelerating computationally intensive time critical AI applications in an energy-efficient manner [14,30]. Cloud platforms are increasingly being augmented with FPGAs to accelerate computation with offering such as Amazon EC2 F1, Microsoft Catapult, Alibaba Faas, etc. The versatility of FPGAs is evident from their widespread deployment in high performance cloud and data-centre platforms [30] as well as in low-powered edge applications [12].

In this work, we develop FASTHash: FPGA-based High Throughput Parallel Hash Table. FASTHash supports p queries ($p > 1$) in each clock cycle, where p is the number of parallel Processing Engines (PEs). To enable such an implementation, we exploit the fact that AI applications are approximate in nature and can tolerate small errors in observations or computations. Thus, we allow the semantic of relaxed eventual consistency i.e. a query inserted by a PE is visible to all the other PEs with a maximum delay of $O(p)$ clock cycles and provide worst case bounds on the erroneous queries (true negative search and duplicate insertion). We implement our hash table entirely using FPGA on-chip SRAM. On-chip SRAM has a very low access latency (1 cycle) compared to

external memory (range of 10 s of cycles). Extremely high bandwidth of up to 38 TB/s is supported by state-of-the-art FPGA devices [30]. Moreover, the abundant on-chip SRAM allows implementation of hash table with entries ranging from several hundred thousands to more than a million.

The key contributions of our paper are:

- To the best of our knowledge, we develop the first "truly" parallel implementation of a hash table on FPGA which supports p operations ($p > 1$) in each clock cycle, thus achieving a throughput of p per clock cycle. The parallel queries in each clock cycle can be any combination of search and insert.
- To fully utilize the abundant low latency on-chip SRAM, we develop novel data organization and query flow techniques. Our techniques allow each of the p PEs to perform hash table search and insert without memory conflicts.
- Our hash table uses relaxed eventual consistency model, i.e. an element inserted from a PE is visible to all the other PEs with some latency. We provide theoretical worst case bounds on the number of queries that are incorrectly served (true negative search or duplicate inserts) due to the relaxed eventual consistency semantics of our hash table.
- Our architecture is flexible and device agnostic. We implement both static and dynamic hash tables on state-of-the-art Xilinx and Intel FPGAs.
- Our hash table designs are scalable to 16 PEs with the static hash table reaching a throughput of 5360 million operations per second at 335 MHz, and the dynamic hash table achieving a throughput of 4480 million operations per second at 280 MHz. They outperform state-of-the-art implementations by 5.7x and 8.7x respectively.

2 Related Work

2.1 Hash Table Implementation on CPU and GPU

Many parallel hashing approaches have been proposed on CPU and GPU platforms. On CPU, significant effort has focused on designing concurrent and lock-free hash table through shared memory and message passing [19,20,24]. With the emergence of many-core architectures, several researches have investigated hash table implementations on GPU [1,9,16]. Essentially, these works divide a hash table into partitions, either coarse or fine grained, and extract parallelism by processing queries to each partition concurrently. In the event that all the queries go to the same partition, they are intrinsically serialized. Recent work by Shankar et al. [25] investigated accelerating Cuckoo hash table using modern CPUs' SIMD instructions, such as Intel AVX2 and AVX-512 extensions. However, their study is limited to lookups, and the complexity incurred by simultaneous lookups and insertions is not considered.

2.2 Hash Table Implementation on FPGA

A number of FPGA-based high performance hash table implementations have been proposed in the community. Among these works, Bando et al. [2] proposed

a hash table based IP lookup technique. Their architecture achieves a lookup throughput of 250 Mops/s. Istvn et al. [15] described a pipelined hash table on FPGA for MemcacheD applications that sustain 10 Gbps throughput. To reduce unnecessary hash table accesses, Cho et al. developed an efficient hash table implementation with bloom filter [6]. Tong et al. [26] developed a data forwarding unit to overcome the data hazards during dynamic hash table updates. Their proposed architecture achieves up to 85 Gbps throughput. Cuckoo hashing implementation of [29] is based on an efficient memory layout. They incorporate a decoupled key-value storage that enables parallel computation of hash values with low overhead. However, all the above works focus on improving performance for a single processing pipeline, which is not sufficient to fully exploit the high bandwidth on-chip SRAM in state-of-the-art FPGAs.

Pontarelli et al. presented an FPGA-based Cuckoo hash table with multiple parallel pipelines [22]. To increase throughput, each pipeline has a different entry point, each of which corresponds to a different hash function. Therefore, the parallelism of their design is limited by the number of hashing functions in a given Cuckoo hash table. Furthermore, due to access conflicts to the same hash function, the achieved throughput with 4 parallel pipelines is only 1.6 queries per clock cycle.

2.3 Novelty of Our Work

State-of-the-art works improve the throughput of hash table by one of the following three techniques: (i) pipelining the implementation to increase the clock frequency, (ii) parallel atomic access to a shared hash table, and (iii) partitioning of hash table to enable parallel access. Technique (i) while improving throughput is clearly not a parallel implementation. Techniques (ii) and (iii) lead to high parallelism if the parallel queries do not need atomic access to the same portion of the hash table or if they map to different partition of the hash table. However, this is highly data dependent and in the worst case all the parallel queries will be serialized leading to reduced throughput similar to a sequential implementation.

In contrast, our implementation processes p parallel queries in each clock cycle, with p being the degree of parallelism. Our implementation is data independent and supports any combination of parallel searches and inserts.

3 Hash Table Overview

3.1 Definition of Hash Table

Hash table is a fast and compact data structure that stores a set S of keys from a much larger universe U. Usually the size of S is much smaller than the size of U. Hash function is used to perform hash table lookup operations. Assume the size of hash table is M, which is usually in the same order as $|S|$, a hash function $h, h : U \rightarrow \{0, ..., M-1\}$, maps a key k in U to an index of M.

The hash table operations supported by our design are:

- 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): $S \leftarrow S \cup \{k, v\}$. Insert a new key-value pair to the hash table if the key does not exist in the hash table at the time of insertion.

There are two forms of hash table, *Static Hash table* and *Dynamic Hash table*. We briefly explain the concept below.

Static Hash Table. In a static hash table, S is a fixed priori, and is immutable during runtime. Therefore, the only operation allowed is search. Perfect hashing is one of the methods to construct static hash tables without collision [7]. One method to construct such hash table using perfect hashing is by employing two levels of hash functions. The first level hash table, in this method, is created using a hash function from the universal hash function family H [5]. For each bucket that has more than 1 item in the first level hash table, i.e. producing some collisions, it creates a second level hash table with $O(n_i^2)$ entries, where n_i is the number of items in bucket i ($n_i > 1$). The second level hash function is also randomly selected from H. A hashing constructed using this method does not have collisions.

Fig. 1. A hash table that is capable of processing p operations in parallel.

Dynamic Hash Table. As its name suggests, dynamic hashing allows search operations while data is incrementally added to a hash table. As a result, dynamic hash table is a mutable object. When a new key is inserted but its respective entry is already occupied, collision is said to occur. On FPGA, collision is usually handled through multiple level of hash functions or linear chaining [15,17,29].

3.2 Parallel Hash Table

In the context of parallel hash table, instead of searching or inserting one key at a time, each query can contain p ($p > 1$) independent operations. The p operations can be in any combination of search and insert. Figure 1 shows the high level concept of parallel hash table. In this case, hash table can complete at most p operations per clock cycle. Designing a parallel architecture on FPGA to efficiently access hash table is a challenging research problem. The primary issue

is resource contentions such as on-chip memory conflicts between concurrent hash table operations. Our proposed architecture guarantees p operations per clock cycle.

4 FASTHash: An FPGA-Based Parallel Hash Table

In this section, we first introduce the novel data organization and query flow in our proposed architecture, to allow concurrent accesses with mixed operation types, and efficiently scale to p parallel processing engines on FPGA. Then we show the architecture details of our design. We also present the extensions to support static hash tables.

4.1 Hash Table Data Organization

Our design goal is to support p hash table accesses per clock cycle with p processing engines. To achieve the desired target, we require a data organization that can perform p parallel operations without stalling.

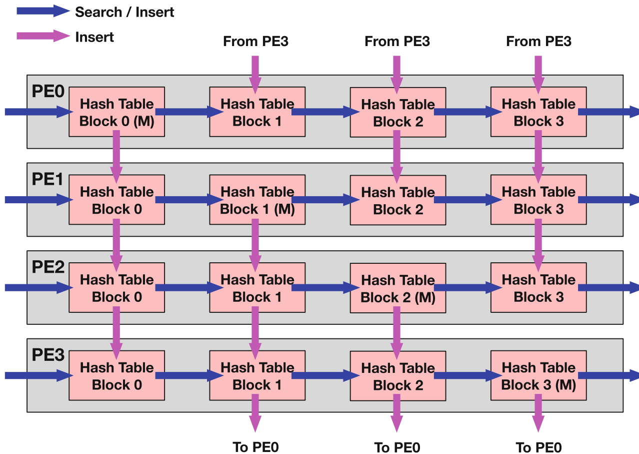


Fig. 2. Data organization and high level architecture of a 4 PEs design. "(M)" indicates the Master Hash Table Block from which a PE initiates insert operation.

The proposed hash table architecture is implemented using on-chip SRAM (BRAM, URAM, or M20K), which is an abundant resource in modern FPGAs [14,30]. Since such memory block is dual-ported, and supports one read and one write per clock cycle, implementing a hash table that guarantees p operations per clock cycle is challenging.

In our proposed design, we assign a copy of the hash table content to each PE. Inside each PE, the hash table is further split into multiple Hash Table Blocks to enable concurrent hash table insert operations. Each Hash Table Block

is mapped to one or more BRAM, URAM, or M20K blocks. To ensure data consistency across PEs, we design an efficient and conflict-free *Inter-PE Dataflow* that connects each Hash Table Block across PEs. With the *Inter-PE Dataflow*, an insert that is made by one PE is visible to all the other PEs with up to O(p) clock cycle delays.

Figure 2 shows an example of our proposed hash table data organization with 4 PEs. Each row represents a PE and the *Inter-PE Dataflow* connects Hash Table Blocks that are in the same column. With this data organization, hash table operations can be performed independently by each processing engine without any memory conflict.

Query Flow. Our hash table architecture supports search and insert operations defined in Sect. 3.2. Before we discuss the query flow of the supported operations, we need to introduce an auxiliary structure called *Master Hash Table Block*.

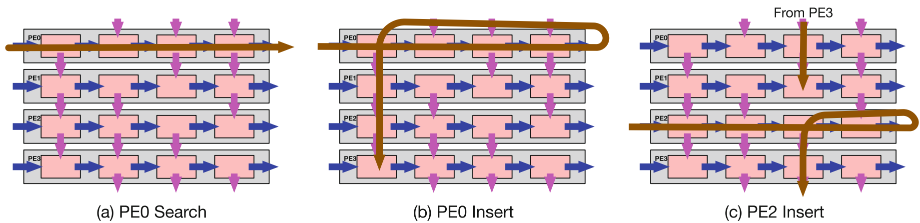


Fig. 3. Query flow in a 4 PEs design.

Master Hash Table Block (MHTB): As we mentioned earlier, we store our hash table in FPGA on-chip SRAM. The split data organization increases the parallelism of our hash table, but it inevitably poses challenges when performing insert to the hash table. This is because we need to keep all the Hash Table Block synchronized in order to return consistent queries. Essentially, an all-to-all communication between PEs would be required during an insert event. To reduce the wiring overhead, we assign block i to PE i – if PE i receives an insert request for a key that does not already exist, it will be inserted by PE i in Hash Table Block i, and subsequently to the whole column i to enable concurrent reads. We refer this Hash Table Block as *Master Hash Table Block (MHTB)*, as shown in Fig. 2. This design guarantees that at any given clock cycle there can only be one insert for any Hash Table Block.

The query flow of our hash table, i.e. mapping of search and insert operations to our parallel architecture, is described below:

Search: Searching for a specific key in our hash table is similar to traditional hash table. Once a PE receives a search query, it goes through a *Lookup Pipeline* that sequentially looks for the key in each Hash Table Block inside the PE. If input key is found, search query flow returns the key-value pair back to application. If no matching key is found, we return empty.

Insert: To insert a key-value pair into our hash table, the operation needs to enter *Lookup Pipeline* first. This ensures the uniqueness of keys in our hash table. If the input key is not found, PE sends the request to *Insert Pipeline* in its MHTB. *Insert Pipeline* connects the MHTB in one PE to non-Master Hash Table Blocks in the other PEs. They receive the same to-be-inserted key-value pair as data flows through the *Insert Pipeline*. Collision is handled by reserving multiple slots for each hash table entry. Figure 3 shows an example of search and insert operations' data flow in a 4 PEs design. As depicted in Fig. 3(b) and Fig. 3(c), upon a simultaneous insert from PE0 and PE2, insert operations are performed by writing to different Hash Table Blocks (columns), thus they never introduce memory conflicts.

Remarks on Supporting Other Hash Table Operations. We focus on a high throughput implementation of hash table suitable for AI applications. As search and insert are the two key operations in such applications, our design is optimized for the same. However, our hash table can be extended to support update (new value for an already inserted key) and delete (remove a key-value pair) operations with little modification by re-routing such operations to the PE which receives the original insert for the corresponding input key. In this work, we make no extra effort to support this feature.

4.2 Hash Table Architecture

Design Overview. As shown in Fig. 2, our proposed design consists of p processing engines (PEs). Each PE can receive input queries independently and with different operation types. Search operations can be completed within each PE itself. For insert operations, each PE needs to propagate changes to the hash table of the other PEs to ensure data consistency. This is achieved by the *Inter-PE Dataflow*. Enforced by our architecture model, inserts initiated by one PE never intervene with inserts by other PEs. Therefore, we can guarantee p parallel accesses per clock cycle in our design.

Processing Engine Design. Figure 4 shows the architecture of our PE design. It contains three key components: *Hashing Unit*, *Data Processing Unit*, and *Collision Handling Unit*. When an input query arrives, it is sent to the *Hashing Unit* to compute the entry index for each hash table block to lookup. *Data Processing Unit* receives the output from *Hashing Unit*. It performs hash table lookup by reading each hash table block sequentially, keeps track of metadata information, and initiates hash table insert if necessary. Both *Hashing Unit* and *Data Processing Unit* are pipelined in order to achieve high operating frequency on FPGA.

Hashing Unit. Hash functions from the Class H_3 [5] has been demonstrated to be effective on distributing keys randomly among hash table entries. The hash function is defined as follows [23]:

Definition 1. Let i denote the number of bits for input key, and j denote the number of bits for hash index. Furthermore, let Q denote a $i \times j$ Boolean matrix. For a given $q \in Q$, let $q(m)$ be the bit string of the mth row of Q, and let $x(m)$ denote the mth bit of input key. The hash function is: $h(x) = (x(1) \cdot q(1)) \oplus (x(2) \cdot q(2)) \oplus ... \oplus (x(i) \cdot q(i))$.

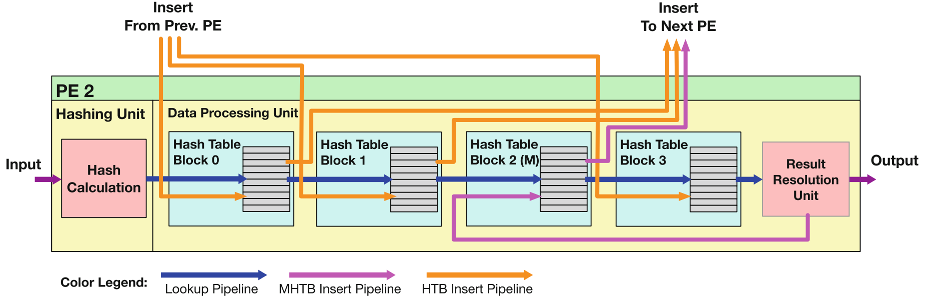


Fig. 4. Architecture of PE 2 in a 4 PEs design. Master Hash Table Block (MHTB) ID is 2 in this case.

We map the hash calculation into a 2-stage pipeline. The first stage simultaneously calculates the AND results for each bit of the input key. The second stage calculates the final hash value by XORing the results of the first stage. Therefore, this hash function calculation logic can be achieved with O(1) latency.

Data Processing Unit. The Data Processing Unit (DPU) handles operations in the order they arrive. As seen in Fig. 4, every operation goes through the *Lookup Pipeline* first; depending on the operation type and lookup result, only insert operations are required to go through *Insert Pipeline*. The entire *Lookup Pipeline* is divided into p mega-stages, as illustrated in Fig. 5(a). Each mega-stage i performs a read operation from hash table block i. Mega-stage can take

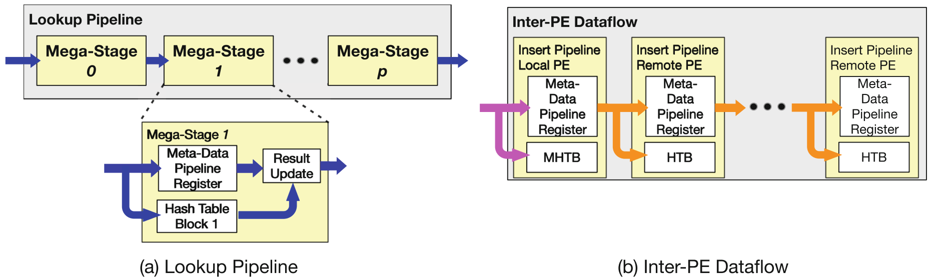


Fig. 5. Implementation details of *Lookup Pipeline* and *Inter-PE Dataflow*.

multiple clock cycles and is pipelined as well. DPU has shift-registers for meta-data information that is needed in a later stage or at the end of the *Lookup Pipeline*. When a key in the hash table is found equal to the input key and the entry is valid, a matching flag is captured and stored into the metadata shift-register.

Result Resolution Unit collects the metadata information and result from the last mega-stage of *Lookup Pipeline*, and routes operations to their next hop. For search operation, it generates the response based on whether a key exists or not. When insert is performed, this unit also inspects the matching flag. If hash table insert condition is satisfied, i.e. matching flag indicates the input key is unique, it issues the operation to the *Insert Pipeline* with metadata information such as hash index, *slot ID*, etc. Otherwise, it generates a response to application indicating failure.

Each Hash Table Block has an *Insert Pipeline*. It is triggered when it receives an operation along with the corresponding metadata information from Result Resolution Unit or from another PE. It writes the new key-value pair and valid information directly to its Hash Table Block. The last stage of *Insert Pipeline* sends the update data to the next PE according to the rules of Inter-PE Dataflow, which we will describe below.

Collision Handling Unit. To handle collision, we design our hash table entry to have multiple slots. Each slot can be allocated to store one key-value pair. One valid field is associated with each slot to indicate if this slot has valid data or available for insertion. An operation is performed only when both the key matches and validity of the slot.

In each PE, only DPU MHTB mega-stage has extra collision handling logic. Other Hash Table Block mega-stage doesn't need collision handling because collision, if any, has already been resolved by the PE which initiates insert operations. Collision is handled by finding the first available slot to insert. We implement a parallel collision handling unit, as shown in Fig. 6. That is: we examine all the slots from a hash table entry at the same clock cycle. This collision handling logic is an extension on top of the hit/miss detection logic that already presents in each Hash Table Block. It has s parallel comparators to detect a matching key.

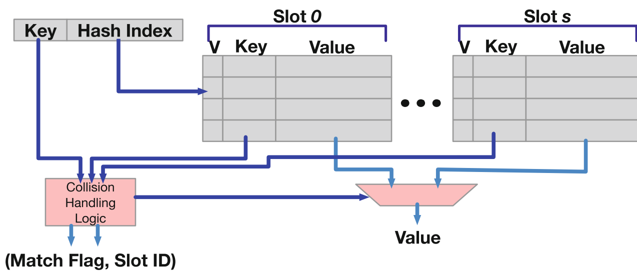


Fig. 6. Parallel collision handling with s slots per Entry. "Hit/Miss/Collision Handling Logic" outputs the outcome of lookup, and *slot ID* for MHTB based on operation type.

The *slot ID* from this stage needs to be recorded into the shift-register because this information is needed by the *Insert Pipeline* later on. Given our low collision rate with H_3 hash functions, we expect 2 to 4 slots per entry to be sufficient. Therefore, it can produce *slot ID* for insert with O(1) latency.

Inter-PE Dataflow. Figure 5(b) shows the Inter-PE communication flow. It plays a vital role in our design to ensure conflict free hash table updates, as discussed in Sect. 4.1. *Inter-PE Dataflow* connects *Insert Pipelines* that are in the neighbor PEs into a "relay network". Therefore, there are totally p parallel *Inter-PE Dataflow* in our architecture. For each *Inter-PE Dataflow*, there is only one *Insert Pipeline* which is capable of initiating hash table insert operations, all the other *Insert Pipelines* in the same "relay network" simply processes the insert and passes the data to the next one, until it reaches the end.

Relaxed Eventual Consistency. As described above, a new key is not seen by all the PEs for up to $pt_0 + p_0 + t_0$ clock cycles, where t_0 is read/write latency of one Hash Table Block. This includes pt_0 clock cycles to search and then $p + t_0$ clock cycles to insert to all the PEs (instead of pt_0 due to pipelining). When the same key is referenced during this time window, our design doesn't make extra effort to forward the data. However, our design guarantees that eventually all accesses to that key will return the inserted value. We refer this behavior as relaxed eventual consistency.

4.3 Customization for Static Hash Table

In order to support perfect hashing with two levels of hash tables, another *Hashing Unit* is added to each PE for the second level hash table. This *Hashing Unit* is placed between the Hashing Unit for the first level hash table and the *Data Processing Unit*. Inside this unit, we use a lookup table to store the hashing functions for each entry in the first level hash table. *Collision Handling Unit* and *Inter-PE Dataflow* are removed because they are designed for insert operations.

5 Hash Table Guarantees and Applications Supported

5.1 Implications of Relaxed Eventual Consistency

An error due to relaxed eventual consistency may occur when the following hold simultaneously: (i) an insert request for a key u is received for the first time; and (ii) another request of search or insert for the same key u is received within $pt_0 + p + t_0$ cycles. Since, every clock cycle serves p requests, we can bound this error by finding number of such issues within $p^2t_0 + p^2 + pt_0$ requests in the sequence of all requests. Note that it is possible to create large number of such errors by having a new key inserted and searched in every clock cycle. However, such cases are unusual in practical setting. Instead, we will assume that there is

a sequence of requests to be served, where the requests (search/insert) occur in small chunks of b keys. The keys in two distinct chunks may be dependent, but all keys within one chunk follow

$$P(\text{one more occurrence of } u | u \text{ has occurred}) \leq P(\text{occurrence of } u) \quad (1)$$

Note that this condition is satisfied, if all the keys within one chunk are independent. For instance, when sampling a sub-graph through edge sampling, we pick edges and hash their vertices. In that case, out of the edges in $\{(u_1, v_1), (u2, v_2), \ldots (u_b, v_b)\}$, $\{u_1, u_2, \ldots, u_b\}$ are mutually independent, and $\{v_1, v_2, \ldots, v_b\}$ are mutually independent. Further, if b vertices coming from $b/2$ edges were considered in the same chunk, they also satisfy the condition as occurrence of a vertex can only reduce the probability of it being selected again (in absence of self loops). Similarly, picking vertices through b independent random walks ensures that vertices within the same chunk are mutually independent. For simplicity, we will pick $b = p^2 t_0 + p^2 + pt_0$.

Theorem 1. *Number of requests n_{err} that are incorrectly served due to relaxed eventual consistency is given by $P(n_{err} \geq \theta) \leq \frac{p^2 t_0 + p^2 + pt_0}{\theta}$.*

Proof. As noted above, we can bound the number of errors, by counting for each key u, the number of times a request for u is made in the same chunk after the first request for u.

Let $C_{i,j}$ be the event that u is requested for the first time in chunk i and the first occurrence is at position j in the chunk. Let n be the total number of requests. Let X_u^f be the number of times a request for u is made within the same chunk just after its first request. Let $X_{u,k}$ be the indicator function for occurrence of u at position k in a chunk. Then, by linearity of expectation:

$$\mathbb{E}(X_u^f) = \sum_{i=1}^{n/b-1} \sum_{j=1}^{b} \left(P(C_{i,j}) \sum_{k=j+1}^{b} \mathbb{E}(X_{u,k} | X_{u,j} = 1) \right) \quad (2)$$

$$\leq \sum_{i=1}^{n/b-1} \sum_{j=1}^{b} \left(P(C_{i,j}) \sum_{k=j+1}^{b} \mathbb{E}(X_{u,k}) \right) \quad (3)$$

$$\leq \left(\sum_{i=1}^{n/b} \sum_{j} P(C_i) \right) \left(\sum_{j=1}^{b} P(X_{u,j} = 1) \right) \leq \sum_{j=1}^{b} P(X_{u,j} = 1). \quad (4)$$

Now, $n_{err} = \sum_u X_u^f$. Therefore,

$$\mathbb{E}(n_{err}) = \sum_u \mathbb{E}(X_u^f) \leq \sum_u \sum_{j=1}^{b} P(X_{u,j} = 1) = b. \quad (5)$$

Markov Inequality leads to

$$P(n_{err} \geq \theta) \leq b/\theta = \frac{p^2 t_0 + p^2 + p t_0}{\theta}. \tag{6}$$

□

5.2 Applications Supported

Hash table is a widely used data structure in various AI algorithms. We list two examples of the AI algorithms and the required hash table characteristics below:

- **Graph Convolutional Neural Network (GCN):** GCN is the generalization of the CNNs to high-dimensional irregular domains represented as Graphs [32]. To tractably handle large graphs, graph sampling is performed to obtain smaller (10,000–50,000 vertices) representative graphs for training. Hashing is used in graph sampling to keep track of the sampled vertices at any given time. With relaxed eventual consistency, it is possible the that same node is sampled multiple times. This will result in an incorrect counting of total nodes sampled. However, this discrepancy is bounded by Theorem 1 and have no effect on subgraph-based graph embedding [33]. In other graph embedding methods that sample neighboring nodes for a given node, such as GraphSAGE [11], this scenario cannot arise because each neighbor is presented only once. **Hash Table Characteristics** [32]: Type: Dynamic. Key size: 32 bits. Value size: 32 bits. Hash Table size: 10,000–50,000.
- **Approximate Nearest Neighbor (ANN) Search:** Given a query point, the objective is to find the point in the dataset closest to the query. Hashing based ANN has been widely adopted in large scale image retrieval [28]. A hash table is created using each sample in the dataset before performing any lookups. Since each point is unique, it is seen only once, relaxed eventual consistency has no effect on the correctness. **Hash Table Characteristics** [28]: Type: Static. Key size: 32–128 bits. Value size: 64 bit (assuming value is memory location of the image). Hash Table Size: 10,000–100,000 entries (equal to image dataset size).

Hash tables are also used for linear function approximation in Reinforcement Learning [10], association rule mining [13], neural network classification [27], etc.

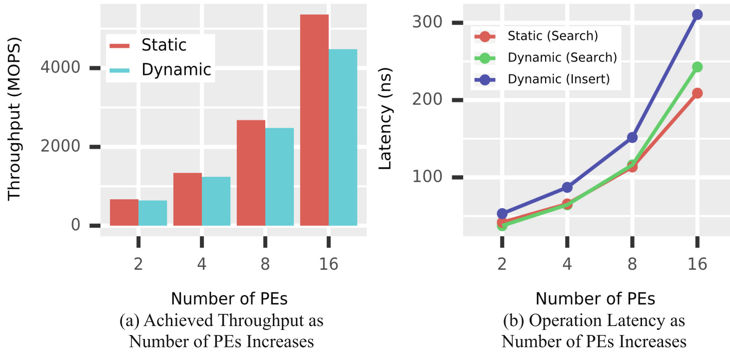
6 Experiments and Results

6.1 Experimental Methodology

We implemented both the static and the dynamic hash tables on Xilinx Alveo U250 FPGA [30] and Intel Stratix 10 MX2100 FPGA [14] using Verilog HDL. The Xilinx device has 1,728,000 LUTs, 3,456,000 Flip-flops, and 327 MB of URAM memory, while the Intel device has 702,720 ALMs, 2,810,880 ALM registers, and 134 MB of M20K memory. Post place-and-route simulations were

performed using Xilinx Vivado Design Suite 2018.3 and Intel Quartus Prime 19.3 respectively. The static hash table, with two levels of hash functions, was created offline using synthetic data.

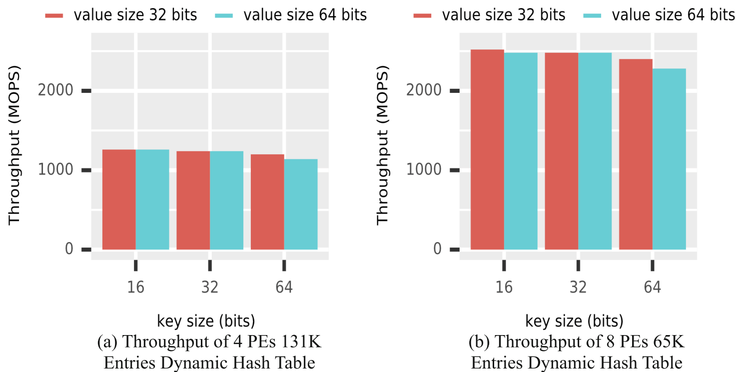
We conducted detailed analysis on the performance, power, and scalability of the proposed architecture. We evaluated the performance and resource utilization by increasing number of PEs from 2 to 16, and varying the total number of hash table entries. The key sizes we used in our experiments were 16, 32, and 64 bits; and value sizes were 32 and 64 bits. These numbers cover the most configurations in AI applications (Sect. 5.2), and they also cover a sufficiently wide range to test the scalability of our architecture. We generated uniformly distributed access patterns, which include both the operation types and the hash keys, as our stimulus. The metric for throughput analysis is million operations per second (MOPS). The utilization of FPGA resources is reported in terms of percent usage of LUTs (ALMs), flip-flops (registers), and on-chip SRAM. To reduce the extra-long duration of the post-route simulation, we used the



(a) Achieved Throughput as Number of PEs Increases

(b) Operation Latency as Number of PEs Increases

Fig. 7. Evaluation of static and dynamic hash tables on Xilinx U250 FPGA. Parameters: key/value length: 32-bit. 4 slots/entry (dynamic hash table).



(a) Throughput of 4 PEs 131K Entries Dynamic Hash Table

(b) Throughput of 8 PEs 65K Entries Dynamic Hash Table

Fig. 8. Throughput with different key and value sizes on Xilinx U250 FPGA.

vectorless power estimation methodology provided by the EDA tools [31]. Power estimation includes leakage and dynamic power.

6.2 Results

Evaluation on Xilinx U250 FPGA. Figure 7 shows the throughput and operation latency of our hash tables from a configuration with 65K entries hash table size. The throughput matches our design goal, which is p operations per clock cycle sustained. The results clearly verify the scalability of our design with the number of PEs. The throughput difference between static and dynamic hashing is due to the max clock frequency. For static hash table scheme, we are able to achieve 335 MHz clock frequency across all PE configurations. On the other hand, parallel collision handling unit and the long wires for inter-PE connections are in the critical paths of our dynamic hashing design. When the number of PEs for dynamic hash table grows, the pressure on place and route also increases. Therefore max clock rate as well as achieved throughput drop when compared with static hashing implementations. Figure 7(b) shows the operation latency increases with the number of PEs. Due to the extra clock cycles that is spent on writing new key-value pair to all PEs, insert latency is higher than search latency. With 16 PEs, search operation can be completed within 209 ns for static hashing and 243 ns for dynamic hashing; insert operation requires 311 ns.

Figure 8 illustrates the throughput as we vary the sizes of hash table keys and values on the dynamic hashing implementation. We find that key and value length have little impact on throughput until the length grows to 64-bit. The clock rate for 4 PEs and 8 PEs configurations drops to 285 MHz and 280 MHz respectively when the size of key and value are both 64-bit. This again demonstrates the scalability of our architecture.

Resource utilization of a dynamic hash table implementation is reported in Table 1. The hash table has 65K entries, 4 slots per entry, and 32-bit key and value length. We make heavy use of URAM for on-chip hash table store. Table 1(a) shows that URAM utilization increases linearly as we increase the number of PEs. This is because each PE stores an entire copy of the hash table. On the other hand, the utilization of other resources, as presented in Table 1(b), is low. Table 1 also shows the estimated power consumption. Our architecture is power efficient, with the power of 16 PEs configuration as low as 9.06 W.

Table 1. Resource utilization of 65K entries dynamic hash table on U250 FPGA.

# of PEs	LUT (%)	Flip-Flop (%)	URAM(%)	Power (W)
2	0.08	0.08	10	3.40
4	0.18	0.31	20	4.13
8	0.66	1.21	40	5.34
16	2.45	4.82	80	9.06

Table 2. Max hash table sizes supported on Xilinx U250 FPGA.

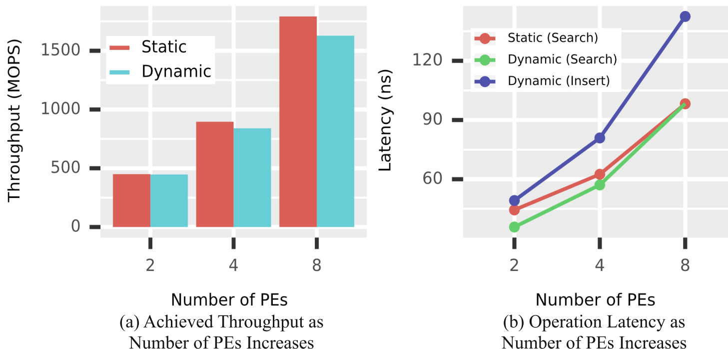
	2 PEs	4 PEs	8 PEs	16 PEs
2 slots per entry	1,310K	655K	327K	131K
4 slots per entry	655K	327K	163K	65K

In Table 2, we show the max hash table size—number of entries that can be implemented on Xilinx U250 FPGA, with 32-bit key and value sizes. Note that as the URAM utilization is pushed to its limit, pipeline depth for each mega-stage in the *Lookup Pipeline* has to be increased to meet optimal timing performance. Our modular design and flexible configurability give user a wide range of design options to choose from based on application requirements and available FPGA resources.

Evaluation on Intel Stratix 10 FPGA. Our architecture is designed as a general solution to work with various FPGA devices. To illustrate, we also implemented our hash table on Intel Stratix 10 FPGA. We used 32-bit as key and value size and used 4 slots per entry for dynamic hash table. Figure 9 shows the performance of our architecture for a hash table with 50K entries. The result

Table 3. Resource utilization of dynamic hash table on Stratix 10 FPGA.

# of PEs	# of entries	ALM (%)	Register (%)	M20K (%)
2	150,000	2	1	59
4	100,000	3	2	78
8	50,000	5	4	81
16	16,000	10	11	52



(a) Achieved Throughput as Number of PEs Increases

(b) Operation Latency as Number of PEs Increases

Fig. 9. Evaluation of static and dynamic hash table on Intel Stratix 10 FPGA. Parameters: key/value length: 32-bit. 4 slots/entry (dynamic hash table).

indicates that the benefits of our architecture is independent of FPGA devices. This design can process up to 1792 MOPS for static hash able and 1628 MOPS for the dynamic version, with 8 PEs. Since the number of PEs affects the max clock rate of the design, the achieved throughput doesn't scale linearly. From Table 3 we can see that the usage of ALM and register is slow, while the utilization of M20K depends on the hash table capacity and number of PEs.

6.3 Comparison with State-of-the-Art (SOTA) Designs

We compare the performance of our 16 PEs hash table implementation on Xilinx U250 FPGA with state-of-the-art GPU and FPGA designs. Performance metric is in term of throughput - MOPS. In [1], the design is implemented on NVIDIA Tesla K40c GPU. The GPU has 2880 CUDA cores. It operates at 745 MHz, and can be boosted up to 876 MHz. The authors report the performance for bulk build (static) and incremental inserts (dynamic) separately. We used 32-bit key/value size and random traffic pattern in the comparison, which is the same as reported by [1,22]. Proposes a parallel Cuckoo hashing on FPGA. Hash table is stored completely on-chip. The target FPGA device is Xilinx Virtex5 XC5VLX155T. Their implementation operates at 156.25 MHz. Table 4 shows the comparison results. Comparing with SOTA GPU work, we observe speedup of 5.7x (static) and 8.7x (dynamic) respectively while running at less than half of the clock rate. Comparing with SOTA FPGA work, our design achieves up to 17x raw speedup, or up to 9.3x speedup after normalizing the clock frequency. Unlike these intrinsically sequential or less optimal parallel implementations, FastHASH fully exploits SOTA FPGA's high bandwidth on-chip SRAM using its unique parallel architecture.

Table 4. Throughput comparison with state-of-the-art (SOTA)

	SOTA GPU [1]	SOTA FPGA [22]	Our Design
Static hashing (search MOPS)	937 (peak)	n/a	5360 (sustained)
Dynamic hashing (MOPS)	512 (peak)	480 (sustained, normalized Fmax)	4480 (sustained)

7 Conclusion

This paper presented FASTHash, a high throughput parallel hash table using FPGA on-chip SRAM that supports p parallel queries per cycle from p PEs ($p > 1$). The architecture is designed to accelerate various AI applications such as graph convolution networks and approximate nearest neighbors. FASTHash uses novel data organization and query flow techniques within and between processing engines to ensure data consistency and conflict-free memory accesses. In

addition, FASTHash is customized to support both static and dynamic hashing on Xilinx and Intel FPGA devices. Both architectures demonstrate high scalability with respect to the number of PEs, key/value lengths, and hash table sizes. The static hash table achieves up to 5360 MOPS throughput and the dynamic variant achieves up to 4480 MOPS, thus outperforming state-of-the-art implementations by 5.7x and 8.7x respectively.

Acknowledgement. This work has been supported by Xilinx and by the U.S. National Science Foundation (NSF) under grants OAC-1911229 and SPX-1919289.

References

1. Ashkiani, S., Farach-Colton, M., Owens, J.D.: A dynamic hash table for the GPU. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 419–429, May 2018
2. Bando, M., Artan, N.S., Chao, H.J.: Flashlook: 100-Gbps hash-tuned route lookup architecture. In: 2009 International Conference on High Performance Switching and Routing, pp. 1–8 (2009)
3. Bengio, Y., et al.: Learning deep architectures for AI. Found. Trends® Mach. Lear. **2**(1), 1–127 (2009)
4. Boulis, C., Ostendorf, M.: Text classification by augmenting the bag-of-words representation with redundancy-compensated bigrams. In: Proceedings of the International Workshop in Feature Selection in Data Mining, pp. 9–16. Citeseer (2005)
5. Carter, J.L., Wegman, M.N.: Universal classes of hash functions (extended abstract). In: Proceedings of the Ninth Annual ACM Symposium on Theory of Computing, STOC 1977, pp. 106–112. ACM, New York (1977)
6. Cho, J.M., Choi, K.: 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, pp. 1–4 (2014)
7. Fredman, M.L., Komlós, J., Szemerédi, E.: Storing a sparse table with 0(1) worst case access time. J. ACM **31**(3), 538–544 (1984)
8. Funge, J.D.: Artificial Intelligence for Computer Games an Introduction. AK Peters/CRC Press, Boca Raton (2004)
9. García, I., Lefebvre, S., Hornus, S., Lasram, A.: Coherent parallel hashing. In: Proceedings of the 2011 SIGGRAPH Asia Conference, SA 2011, pp. 161:1–161:8. ACM, New York (2011). https://doi.org/10.1145/2024156.2024195. http://doi.acm.org/10.1145/2024156.2024195
10. Gendron-Bellemare, M.: Fast, scalable algorithms for reinforcement learning in high dimensional domains (2013)
11. Hamilton, W.L., Ying, R., Leskovec, J.: Inductive representation learning on large graphs. In: Proceedings of the 31st International Conference on Neural Information Processing Systems, NIPS 2017, pp. 1025–1035. Curran Associates Inc., New York (2017)
12. Hao, C., et al.: FPGA/DNN co-design: an efficient design methodology for IoT intelligence on the edge. arXiv preprint arXiv:1904.04421 (2019)

13. Holt, J.D., Chung, S.M.: Mining association rules in text databases using multipass with inverted hashing and pruning. In: Proceedings 14th IEEE International Conference on Tools with Artificial Intelligence 2002, ICTAI 2002, pp. 49–56. IEEE (2002)
14. Intel: Stratix 10 MX FPGAs. https://www.intel.com/content/www/us/en/products/programmable/sip/stratix-10-mx.html
15. István, Z., Alonso, G., Blott, M., Vissers, K.: A flexible hash table design for 10 Gbps key-value stores on FPGAS. In: 2013 23rd International Conference on Field programmable Logic and Applications, pp. 1–8 (2013)
16. Khorasani, F., Belviranli, M.E., Gupta, R., Bhuyan, L.N.: Stadium hashing: scalable and flexible hashing on GPUs. In: 2015 International Conference on Parallel Architecture and Compilation (PACT), pp. 63–74, October 2015
17. Kumar, S., Crowley, P.: Segmented hash: an efficient hash table implementation for high performance networking subsystems. In: Proceedings of the 2005 ACM Symposium on Architecture for Networking and Communications Systems, ANCS 2005, pp. 91–103. ACM, New York (2005)
18. Kuppannagari, S.R., et al.: Energy performance of FPGAs on perfect suite kernels. In: 2014 IEEE High Performance Extreme Computing Conference (HPEC), pp. 1–6. IEEE (2014)
19. Metreveli, Z., Zeldovich, N., Kaashoek, M.F.: CPHash: a cache-partitioned hash table. In: Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2012, pp. 319–320. ACM, New York (2012)
20. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: Proceedings of the Fourteenth Annual ACM Symposium on Parallel Algorithms and Architectures, SPAA 2002, pp. 73–82. ACM, New York (2002)
21. Peters, J.: Machine learning for motor skills in robotics. KI-Künstliche Intelligenz **2008**(4), 41–43 (2008)
22. Pontarelli, S., Reviriego, P., Maestro, J.A.: Parallel d-pipeline: a cuckoo hashing implementation for increased throughput. IEEE Trans. Comput. **65**(1), 326–331 (2016)
23. Ramakrishna, M.V., Fu, E., Bahcekapili, E.: Efficient hardware hashing functions for high performance computers. IEEE Trans. Comput. **46**(12), 1378–1381 (1997)
24. Shalev, O., Shavit, N.: Split-ordered lists: lock-free extensible hash tables. J. ACM **53**(3), 379–405 (2006)
25. Shankar, D., Lu, X., Panda, D.K.: SimdHT-bench: characterizing SIMD-aware hash table designs on emerging CPU architectures. In: 2019 IEEE International Symposium on Workload Characterization (IISWC) (2019)
26. Tong, D., Zhou, S., Prasanna, V.K.: High-throughput online hash table on FPGA. In: 2015 IEEE International Parallel and Distributed Processing Symposium Workshop, pp. 105–112 (2015)
27. Vijayanarasimhan, S., Yagnik, J.: Large-scale classification in neural networks using hashing, US Patent 10,049,305, 14 August 2018
28. Wang, D., et al.: Supervised deep hashing for hierarchical labeled data. In: Thirty-Second AAAI Conference on Artificial Intelligence (2018)
29. Liang, W., Yin, W., Kang, P., Wang, L.: 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), pp. 1–4 (2016)
30. Xilinx: Alveo U250 data center accelerator card. https://www.xilinx.com/products/boards-and-kits/alveo/u250.html

31. Xilinx: Vivado design suite. https://www.xilinx.com/products/design-tools/vivado.html
32. Zeng, H., Zhou, H., Srivastava, A., Kannan, R., Prasanna, V.: Accurate, efficient and scalable graph embedding. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 462–471. IEEE (2019)
33. Zeng, H., Zhou, H., Srivastava, A., Kannan, R., Prasanna, V.K.: GraphSAINT: graph sampling based inductive learning method. CoRR abs/1907.04931 (2019). http://arxiv.org/abs/1907.04931