



A Parallel Hash Table for Streaming Applications

Downloaded from: <https://research.chalmers.se>, 2026-09-02 19:46 UTC

Citation for the original published paper (version of record):

Östgren, M., Sourdis, I. (2024). A Parallel Hash Table for Streaming Applications. Parallel Architectures and Compilation Techniques - Conference Proceedings, PACT: 297-308.
<http://dx.doi.org/10.1145/3656019.3676951>

N.B. When citing this work, cite the original published paper.

© 2024 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, or reuse of any copyrighted component of this work in other works.



A Parallel Hash Table for Streaming Applications

Magnus Östgren
Chalmers University of Technology
Sweden
magnusos@chalmers.se

Ioannis Sourdis
Chalmers University of Technology
Sweden
sourdis@chalmers.se

Abstract

Hash Tables are important data structures for a wide range of data intensive applications in various domains. They offer compact storage for sparse data, but their performance has difficulties to scale with the rapidly increasing volumes of data as they typically offer a single access port. Building a hash table with multiple parallel ports either has an excessive cost in memory resources, i.e., requiring redundant copies of its contents, and/or exhibits a worst case performance of just a single port memory due to bank conflicts. This work introduces a new multi-port hash table design, called Multi Hash Table, which does not require content replication to offer conflict free parallelism. Multi Hash Table avoids conflicts among its parallel banks by (i) supporting different dynamic mappings of its hash table address to index to the banks, and by (ii) caching (and aggregating) accesses to frequently used entries. The Multi Hash Table is used for reconfigurable single sliding window stream aggregation, increasing processing throughput by 7.5×.

CCS Concepts

• **Hardware** → **Hardware accelerators.**

Keywords

FPGA, hash table, high throughput, stream aggregation

ACM Reference Format:

Magnus Östgren and Ioannis Sourdis. 2024. A Parallel Hash Table for Streaming Applications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '24)*, October 14–16, 2024, Long Beach, CA, USA. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/3656019.3676951>

1 Introduction

In an era where massive volumes of data are produced and collected constantly, the value of these data depends on how efficiently and fast they are processed. Hash tables are key data structures for a wide range of data intensive applications in various domains as they provide fast access and compact storage to sparse data. For example, sparse matrix multiplication, used in machine learning [21, 22, 40], graph processing [15, 36], and numerous other algorithms [42], can employ hash tables to merge partial results [31]. In network processing, hash tables are used to store the state of packet flows [25, 32] as well as the search patterns of payload content used in deep packet inspection [8]. Stream processing, data analytics

and other applications that process key-value pairs rely on hash tables to handle data for different keys [6, 10, 11, 23]. Such workloads are expected to process enormous amounts of data at line rates. However, hash tables have difficulties to scale and introduce performance bottlenecks.

Conventional hash tables have limited throughput because they are sequential and allow a single access per cycle. Building a multi-port hash table to support multiple parallel accesses has an excessive cost of resources and/or limited worst case performance. More specifically, memory resources increase at least linearly ($O(N)$) to the number of (read) ports when write accesses are rare and inconsistencies between redundant copies of hash table entries can be tolerated by the application [43, 44]. The memory size increases quadratically ($O(N^2)$) to the number of ports when read and write accesses need to be serviced accurately [27]. On the other hand, a hash table can be split without any redundancy to multiple (N) banks and support N parallel accesses [8], but then bank conflicts can limit worst case performance to that of a single port table. In summary, currently the memory cost of N ports is at least $O(N)$, if not $O(N^2)$, otherwise performance is data dependent and can be reduced to a single port throughput.

This work introduces a new parallel multi-port hash table for streaming applications, which overcomes the above drawbacks of existing approaches and offers data independent parallelism. The proposed design, called Multi Hash Table, is organized in multiple banks, each using a separate queue for incoming access requests. Our design does not rely on redundant content for conflict-free parallelism. On the contrary, each bank stores a disjoint subset of the entries and bank conflicts are avoided based on the following two complementing mechanisms. The first one supports multiple hash functions, in particular multiple bit-arrangements of the same hash function output, and allows to dynamically select and switch to one of them providing alternative address mappings to the table and improving load balance across banks. The second mechanism adds a cache before the banks to merge accesses to frequently accessed entries. It is advocated that the combination of (i) dynamic switching of address mappings and (ii) caching requests to frequently accessed entries can offer data independent multi-port hash table performance without wasting memory resources for replicating hash table contents. Multi Hash Table is applied to a reconfigurable stream aggregation accelerator and improves throughput manifold.

Concisely, the contributions of this paper are the following. A new hash table design is introduced, which maintains data-independent multi-port performance via dynamic address remapping and caching of frequent accesses. A theoretical analysis of the proposed Multi Hash Table is performed to determine the minimum cache size and address mappings for maintaining throughput. A reconfigurable stream aggregation accelerator, which employs the proposed Multi Hash Table increasing processing throughput 7.5×.



This work is licensed under a Creative Commons Attribution-Share Alike International 4.0 License.

PACT '24, October 14–16, 2024, Long Beach, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0631-8/24/10
<https://doi.org/10.1145/3656019.3676951>

The remainder of this paper is organized as follows: Section 2 discusses related work and presents background on stream aggregation. Section 3 offers a theoretical analysis of the proposed Multi Hash Table. Section 4 describes our reconfigurable stream aggregation accelerator with a Multi Hash Table. Section 5 provides our evaluation results and Section 6 summarizes our conclusions.

2 Background & Related Work

This Section describes related work on multi-port memory and hash tables as well as on other relevant approaches and offers background on sliding window stream aggregation systems where the Multi Hash Table concepts are applied.

2.1 Related work

Memory parallelism has been a cornerstone of high performance computing. In the past, many designs for multi-port and multi-bank memories and hash tables have been proposed.

Some approaches opt for replicating memory contents to provide multiple parallel accesses. LaForest and Steffan described a design for multi-port SRAMs in FPGAs [27]. It supports m -write and n -read ports multiplexing m SRAM blocks, each storing one copy of the data and providing one write and n read ports. An additional smaller true m write, n read port memory is used to keep track of the SRAM copy that stores the most recently updated version of each memory entry. The memory cost of the design is therefore $O(m \times n)$ or $O(N^2)$ for providing N read and N write ports. Yang et al. designed a parallel hash table that offers N parallel (read) access ports replicating N times the contents of the table [43, 44]. Write accesses need to be broadcasted to all copies so they are effectively handled with the performance of a single port memory, and without a synchronization mechanism, temporary inconsistencies in the memory contents may occur. In summary, the proposed design offers N parallel reads at the memory cost of $O(N)$ copies, the write accesses are performed at a single port memory throughput and may introduce inaccuracies.

Several designs avoid the replication of hash table contents and use multiple banks coupled with techniques that alleviate bank conflicts. HashCache achieves high speed stateful network processing at an input rate of 200 million packets per second, supporting up to 800 million different flows [32]. It employs a multi-bank hash table to store state per flow and uses a small, fixed size cache to service a handful of frequently appearing packet flows without consuming the bandwidth of the multi-bank hash table. However, the design is vulnerable to any traffic composed of repeated packet flows that do not all fit in their limited cache. A multi-bank hash table is also used for hash-based pattern matching to scan the payload of network packets in a deep packet inspection system [8]. Hash-based pattern matching uses perfect hashing on incoming data to index to a memory that stores search patterns. A subsequent comparison between the read pattern and incoming data confirms the match [8, 39]. Fukac et al. increased the throughput of their hash-based pattern matching design accepting N incoming payload bytes per cycle, hashing them N times at different byte offsets and producing N accesses per cycle to a multi-bank memory that stores the search patterns [8]. Conflict resolution is (only partly) handled by a network that interconnects the outputs of the hash functions and the

banks. The network is able to deduplicate redundant accesses to the same address (search pattern) if these accesses meet in the network. A similar mechanism was designed for the NYU Ultracomputer [16] for reducing the traffic to the multiple memory banks. The Ultracomputer used an omega network and, among others, merged and deduplicated memory requests to the same address. Similar to the Ultracomputer, in the design of Fukac et al., accesses to different addresses that are mapped to the same bank are still not handled and can cause bank conflicts. An interesting technique to reduce bank conflicts in DRAM is the Duplicon cache [29]. It reserves a small part of DRAM to selectively duplicate (in practice cache) the contents of memory locations that cause a large number of conflicts. Multi Hash Table differs from the Duplicon cache because it offers alternative address mappings to store data, which would be inefficient in the Duplicon cache context. It is also different from the Ultracomputer [16] and the hash-based pattern matching by Fukac et al. [8] because it uses alternative address mappings and caching to avoid bank conflicts rather than just only deduplicating multiple accesses to the same address.

Another work that is related to the use case of the proposed Multi Hash Table in a key-value pair stream aggregation system is the sort-reduce technique in GraFBoost [24]. GraFBoost proposes a merge-sort followed by a reduce operation to reduce the number of incoming key-value pairs in graph analytics and alleviate the bandwidth pressure. In our stream aggregation Multi Hash Table, a similar mechanism is used to sort and merge key-value pairs (tuples) that belong to the same key before caching as well as to merge tuples within the cache. However, as opposed to GraFBoost, our stream aggregation use case targets non-associative functions, so multiple values of the same key cannot be reduced to a single value, hence our merging step produces a larger tuple with a key and all aggregated values of the merged tuples.

The use of multiple alternative address mappings for reducing bank collisions in Multi Hash Table can be generalized as use of multiple hash functions and is therefore related to existing approaches that employ multiple hash functions. Seznec's skewed associative cache uses a different hash function for each way (bank) of a cache to reduce (hash-) collisions within a set and improve cache utilization [38]. Cuckoo hashing does something similar to reduce hash-collisions handling insertions differently, i.e., inserting a new entry (to one bank) with one hash function and attempting to re-insert the evicted entry (to another bank) with another hash function [34]. An interesting hardware implementation of Cuckoo hashing exploits the fact that incoming requests do not always use all hash functions and do not access all banks [35]. Capitalizing on this observation, it increases throughput by $1.6\times$ with a pipeline for bank accesses, which allows requests to enter at any idle bank stage [35]. However, both skewed caches and Cuckoo hashing aim to improve the utilization of hash table capacity reducing hash-collisions, i.e., the collisions of multiple keys to the same hash table entry. On the contrary, the aim of our alternative address mappings approach is to reduce the collisions of keys to the same memory bank rather than to a single memory entry, thereby increasing throughput, rather than improving capacity utilization. In fact, Multi Hash Table can be orthogonal to the choice of hash function and its efficiency in reducing hash collisions because it only affects the address mapping of the hash function output. Moreover,

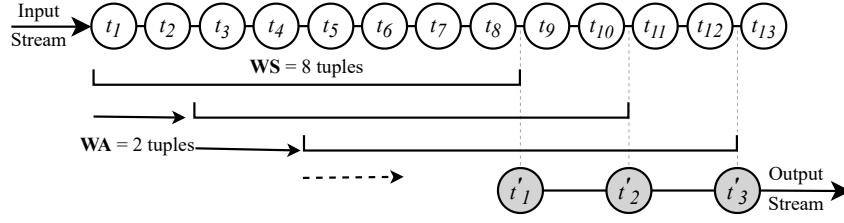


Figure 1: Sliding-window stream aggregation with Window Size (WS)=8 tuples and Window Advance (WA)=2 tuples for an input stream of key-value pair tuples $t_1, t_2, \dots$. Grey tuples, t'_1, t'_2, t'_3 , indicate output of an aggregation function generated based on the sliding-window contents when window gets full.

lookups in skewed caches and Cuckoo hashing would require multiple (possibly parallel) accesses to multiple banks, which would be wasteful in terms of throughput. On the other hand, Multi Hash Table lookups use only the current address mapping unless a new key is inserted, in which case locations of alternative mappings need to be accessed, too, to ensure the key is not already stored in the table.

2.2 Stream aggregation with reconfigurable acceleration

Stream aggregation is one of the most challenging tasks in stream processing. It can be described by applying the traditional relational database aggregation semantics to a sliding window. However, as opposed to databases, it is used to analyze *unbounded* streams of big data in various domains, e.g., financial, transportation. Such a sliding window of size WS elements is updated with new incoming elements (values carried by incoming tuples) as illustrated in Figure 1. Upon aggregation, the window “slides” by a particular number of elements (Window Advance - WA) to produce the aggregated values, i.e., the window contents before sliding [4, 9]. The aggregated values are subsequently fed to one or multiple functions that compute an output every time the window slides. Considering a key-value pair system, incoming tuples carry values of different keys, which are aggregated separately maintaining a separate sliding-window per key. Each incoming tuple uses its key as input to a hash function to index to a hash table that stores data per key, i.e., the values aggregated in the sliding window and metadata needed to handle the window.

For some problems, the sliding-window aggregations can be simplified by computing them incrementally [28, 30, 33]. However, many others need to follow the single sliding window stream aggregation (Single-SWAG) approach [10, 11, 13]. That is the case for problems that use non-associative, holistic aggregation functions, which cannot be computed incrementally, e.g., median [17], or problems that would be more expensive to compute incrementally than using Single-SWAG, e.g., frequent aggregations of multiple aggregation functions in geo-tagged data [20], social-media data [19] or manufacturing-equipment data [18].

Reconfigurable hardware is a suitable substrate for accelerating stream aggregation because it offers hardware parallelism and the opportunity to customize the design to particular WS and WA as well as to the aggregation functions needed by the application.

Despite previous efforts to accelerate single window stream aggregation using reconfigurable hardware [10, 11] with designs that offered among others specialized memory hierarchies [12], or compressed sliding windows [13, 14], existing solutions use conventional hash tables that support one read and one write access per cycle to offer a single read-modify-write operation per incoming tuple [23]. This effectively limits the throughput of previous stream aggregation designs to, at best, one incoming tuple per cycle. As shown in Section 4, the proposed Multi Hash Table enables the processing of multiple incoming tuples per cycle, increasing the throughput of reconfigurable sliding window stream aggregation.

3 Theoretical analysis of the Multi Hash Table

This Section provides a theoretical analysis of the proposed scheme and its two mechanisms for avoiding conflicts independently of the distribution of incoming access requests. The main objective of this analysis is to determine the number of cache entries C that is sufficient to avoid bank conflicts and maintain maximum throughput of $N = 2^n$ parallel accesses per cycle. Let us consider the generic, abstract view of the Multi Hash Table illustrated in Figure 2, with N parallel incoming accesses per cycle. The hash table is split to $B = 2^b$ banks of $S = 2^s$ entries each, where $B \geq N$ in order to support N accesses per cycle. Each incoming request uses a key to hash to $a = b + s$ bits, which are used as the address to index to the hash table of 2^a entries. Up to N parallel incoming requests access first an N -port cache, which stores multiple requests for

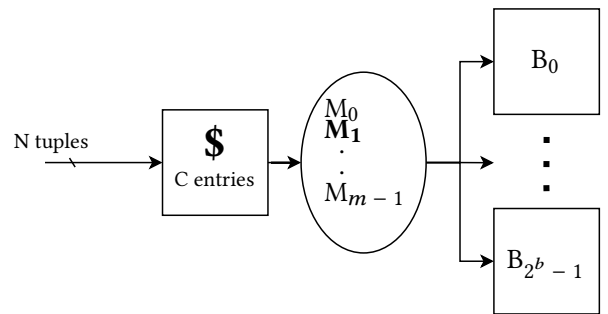


Figure 2: Generic view of the Multi Hash Table. Tuples stream through an access cache, and then to a hash table bank according to the currently active address mapping.

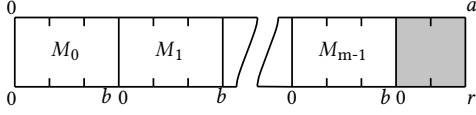


Figure 3: Breakdown of the Multi Hash Table address space.

C unique hash table locations (addresses). Each cache entry can aggregate multiple (up to N) requests to the same hash table address, which can be sent together to the hash table as one access¹. An incoming request that hits in the cache is stored in the cache if it fits. Otherwise, N aggregated requests to the respective hash table address are evicted and sent together to the table as a single hash table access. In case the request misses, then depending on the cache replacement policy it may be either cached, evicting a previous entry, or it may bypass the cache going directly to the hash table banks.

In this analysis, an ideal cache is considered, which always stores the most recently used entries limited only by its capacity. The cache replacement policy prioritizes storing entries of the busiest banks. Moreover, switching to a new address mapping is instant and does not entail any cost for moving hash table entries. These considerations are discussed in the next Section, which describes how and to what extent the proposed Multi Hash Table design is able to support them when applied to a stream aggregation system.

Let us consider that the hash table can switch between m different address mappings, $M_0, M_1, \dots, M_{m-1}$. Each address mapping uses b (out of a) address bits to select a bank. Let us assume there are r address bits, which are not used by any mapping to select a bank. To minimize the number of mappings and maximize the address bits used by the mappings to select a bank, it can be considered that there is no overlap between the b bits used by each mapping to select a bank as shown in Figure 3. Then, the total number of address bits is $a = m * b + r$.

For any address mapping, repeated accesses to a single address would always need to be serviced by the same bank limiting throughput. Such repeated accesses would need to be serviced by the cache in order to avoid congestion to the particular bank and thus performance degradation. Similarly, any sequence of accesses to up to $N - 1$ specific addresses would always go to less than N different banks, reducing the expected throughput. These are some examples that motivate the use of a cache before the hash table banks.

The above address breakdown of Figure 3 can be used to estimate a first bound to the minimum needed cache size $C^{N,m,r}$ as follows: let us consider for each of these $i = 0, 1, 2, \dots, m$ address parts the maximum number of distinct values X_i they can take and still cause bank conflicts. Then, the total number of distinct addresses that can cause bank conflicts and would need to be cached is equal to the product of the number of distinct values for each address part: $\prod_{i=0}^m X_i$.

For a particular mapping M_i , throughput would be limited if the N parallel accesses go to fewer than N banks. That is that the b bits used for selecting a bank have $X_i \leq N - 1$ distinct values accessing an equal number of banks. Similarly, the b bits used for selecting a

bank in the other alternative mappings should also have up to $N - 1$ distinct values. Otherwise, if any alternative mapping has more than that, i.e., $X_i \geq N$, it is beneficial to switch to that alternative mapping and restore balance. Finally, the r bits that are not used by any mapping to select a bank can take up to 2^r values. Based on the above, a first bound to the minimum needed cache size is:

$$C^{N,m,r} = (N - 1)^m * 2^r \quad (1)$$

where the b bits for each mapping take up to $N - 1$ distinct values and the r unused bits take 2^r possible values.

The cache size can be bounded further considering that N parallel incoming requests can conflict in at most $\frac{N}{2}$ banks, because any bank conflict requires at least two requests to that bank. For example, if the N requests map to $N - 1$ banks, then only a single bank conflict is caused and needs to be serviced by the cache. In other words, for the current mapping M_i , the number of banks with conflicts, denoted as K_i , determines the number of distinct values of the corresponding b address bits, which contribute to the unique addresses that need to be stored in the cache. Any other access can go directly to a bank without conflicts. For example, when all $N = 8$ accesses go to bank #1, although all (but one) of these requests need to be handled by the cache, they all have the same value for the corresponding b bits of the address, the value "1". On the other hand, when the $N = 8$ accesses go to five banks, e.g., as follows: #1, #2, #3, #4, #5, #1, #2, #3, respectively, although the b bits of the mapping take five distinct values, only (up to) three of these values will cause conflicts to banks #1, #2, and #3, and therefore only three values of these b bits are part of addresses that need to be cached.

For a mapping M_i , where the N parallel accesses map to X_i banks, the maximum number of conflicting banks is:

$$K_i^{N,X_i} = \min(X_i, N - X_i) \leq \frac{N}{2} \quad (2)$$

For example, for $N = 8$, $K_i^{8,7} = K_i^{8,1} = 1$, $K_i^{8,6} = K_i^{8,2} = 2$, $K_i^{8,3} = K_i^{8,5} = 3$, $K_i^{8,4} = 4$, $K_i^{8,8} = 0$. For the b bits used by the current mapping M_i to select a bank, this defines how many distinct values contribute to the addresses that need to be cached.

Let us consider that the current mapping causes K_i bank conflicts. Then, the maximum number of distinct values X_i should be calculated for the address part used by each alternative mapping in order for it to produce K_i or more conflicting banks. In case an alternative mapping causes fewer conflicting banks, it is considered beneficial to switch to that mapping and repeat the above analysis. For the b bank-selecting bits of a mapping, the maximum number of distinct values X_i that cause K_i or more conflicting banks is:

$$X_i^{N,K_i} = N - K_i \quad (3)$$

e.g., $X_i^{8,2} = 6$ because $X_i = 2$ or 6 causes 2 conflicting banks, $X_i = 3$ or 5 causes 3, and $X_i = 4$ causes the maximum number of conflicting banks $\frac{N}{2} = 4$, so the maximum X_i for $N = 8$ with $K_i = 2$ or more conflicting banks is 6 distinct values. Note that $X_i = 1$ or 7 cause only one conflicting bank and $X_i = 8$ zero conflicts.

Then, for a given number of conflicting banks j , the number of distinct addresses that need to be cached is equal to the product of the number of conflicting banks j (for the current mapping), and the maximum distinct values that cause equal or more conflicting banks $(N - j)$ for each alternative mapping, that is: $j * (N - j)^{m-1}$.

¹Depending on the use of the hash table, multiple requests could be reduced in the cache without accessing the hash table in various ways, e.g., in incremental aggregations [28, 30, 33] or cases similar to the memory accesses of the Ultracomputer[16].

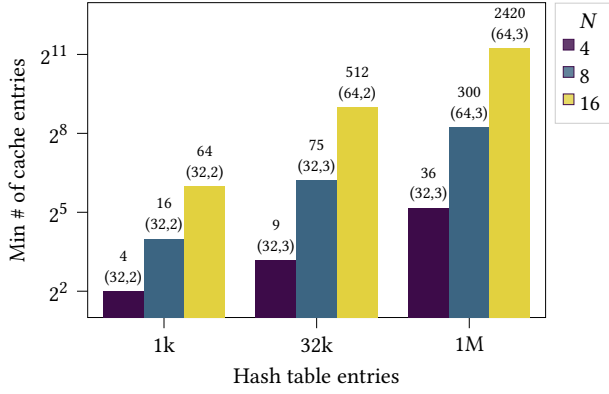


Figure 4: Minimum number of cache entries needed to maintain processing throughput of N incoming tuples per cycle for different sizes of hash tables. At the top of each bar the exact number of cache entries is shown, as well as, in a parenthesis, the number of banks (B) and number of address mappings (m) used in the selected design point (B, m).

The minimum cache size then is equal to the largest such product for any possible number of conflicting banks, multiplied by the 2^r possible values of the remaining r address bits not used by any mapping:

$$C^{N,m,r} = \max_{j=1}^N [j * (N - j)^{m-1}] * 2^r \quad (4)$$

e.g., $C^{8,3,0} \leq \max(4*4^2, 3*5^2, 2*6^2, 1*7^2) = \max(64, 75, 72, 49) = 75$.

Based on the above, for a 32K entry Multi Hash Table ($a = 15$) which accepts $N = 8$ parallel accesses, uses $B = 32$ banks of $S = 1K$ entries each, and supports $m = 3$ non-overlapping address mappings to select a bank using all address bits ($b * m = 15 = a$, so $r = 0$), a cache of minimum size $C^{8,3,0} = 75$ entries is needed to maintain a throughput of 8 accesses per cycle. The above setup without the support of alternative address mappings ($m = 1$), would need a cache of $C^{8,1,0} = 4K$ entries. Similarly, for a 2M entry Multi Hash Table ($a = 21$) which accepts $N = 16$ parallel accesses, uses $B = 128$ banks of $S = 16K$ entries each, and supports $m = 3$ non-overlapping address mappings to select a bank using all a address bits ($b * m = 21 = a$, so $r = 0$), a cache of minimum size $C^{16,3,0} = 605$ entries is needed to maintain a throughput of 16 accesses per cycle. Again, without the support of alternative address mappings, the required cache size would be $C^{16,1,0} = 128K$. Figure 4 shows the minimum cache size for hash tables of 1K, 32K and 1 million entries processing $N = 4, 8$, and 16 tuples per cycle, using $m \leq 3$ address mappings and $B \leq 64$ banks. It can be observed that the required cache size scales well (sub-linearly) to the hash table capacity and quadratic $O(N^2)$ or cubic $O(N^3)$ to the number of incoming tuples per cycle.

4 Multi Hash Table Design for Stream Aggregation

The Multi Hash Table is designed for and used in a reconfigurable system for stream aggregation, as defined in Section 2.2, aiming to increase its processing throughput. Incoming tuples of key-value

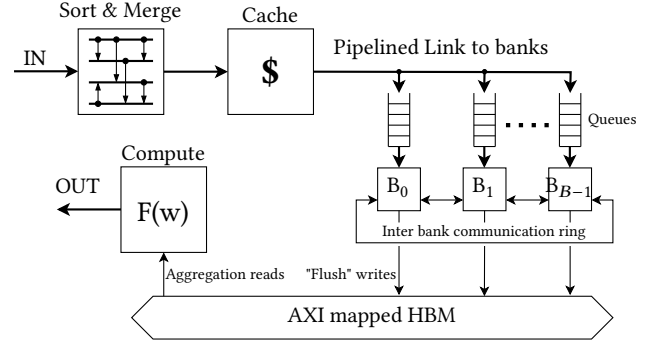


Figure 5: Top-level Block diagram of our Stream Aggregation system with Multi Hash Table.

pairs $\langle k, v \rangle$ are aggregated per key in sliding windows of size WS tuples, which advance/slide by WA tuples to feed with their contents some compute function, e.g., average, max, sum, median. The proposed Multi Hash Table enables the design of a reconfigurable stream aggregation accelerator to process N incoming tuples per cycle, substantially increasing its throughput independently of the incoming key distribution.

Figure 5 illustrates the top-level block diagram of the design. It is composed of a sort-and-merge network, a subsequent cache, a hash table using multiple parallel SRAM banks for window management and partial data aggregation, and a high bandwidth DRAM (HBM) for storing the complete window contents, as well as a compute stage, which produces the final output. N incoming tuples $\langle k, v \rangle$ enter the system simultaneously. The N tuples are first sorted by key and merged if they belong to the same key. Subsequently, they access an N -port cache with their keys. The cache replacement policy prioritizes the caching of accesses that map to banks with busier queues. Cache hits enable the value(s) of the respective incoming tuple to be stored next to older, already cached values of the same key. Each cache entry fits a fixed number of values, and exceeding this number causes the key and its cached values to be evicted. Each cycle, up to N cache accesses produce up to N evictions, which are then sent forward to the next stage. The keys of evicted tuples are hashed considering the current address mapping to produce the hash table address and index to the multi-bank hash table. They subsequently enter the queue of the selected bank and then access the bank with a read-modify-write operation, which utilizes both available bank ports in a pipelined fashion. Each hash table entry stores the key, metadata to manage the sliding window, such as pointers to the DRAM location of the sliding window, the number of values in the window, etc., as well as the most recent incoming values, which are flushed to DRAM in groups that match the granularity of DRAM accesses. Windows ready for aggregation are detected, and their data are sent from the DRAM to the final stage, which computes the aggregation function(s) using one of its multiple parallel units. Control logic is used to handle the dynamic switching between m hash function address mappings based on the occupancy of the bank queues. A ring that connects all banks is used to exchange messages and metadata information that allows the relocation of hash table entries when needed. Upon

relocation, actual data are flushed to HBM, avoiding any costly data movement across banks. Dynamic change of address mapping requires that when a new key is inserted into the hash table, the other $m - 1$ alternative locations of the key need to be checked, too, for an existing entry of the key. However, any other regular hash table lookup (key hit) is performed with a single bank access using the current mapping. Next, each of the above components of the proposed design are described separately.

4.1 Sort-and-Merge

The N parallel incoming tuples are first sorted by key using a pipelined sorting network. This network is implemented using an optimal network [26] when the number of inputs allows for it; otherwise as bitonic merge sort. At this stage, latency is not critical, so fine-grain pipelining can be applied to ensure high clock rates. Subsequently, tuples of the same key are merged into a single multi-value tuple, i.e., $\langle k, v_1, v_2, \dots \rangle$, which aggregates all values of the incoming key. As opposed to previous work on sort-and-reduce [24], here it is considered that the values of a sliding window feed non-associative functions and therefore multiple values of the same key cannot be reduced to a single value before the compute stage. Still, merging in a multi-value tuple allows it to deduplicate tuples of the same key and avoid multiple accesses to the subsequent stages, i.e., cache, banks, using the same key. As shown in the example of Figure 8, in practice, the output of the merging step still uses the same N -tuple input datapath format, but marks the unique keys, which will subsequently access the cache, and the value(s) they carry. At this stage, before accessing the cache, the order of the input tuples are shuffled to improve cache utilisation, because the order of the tuples affects cache replacement as explained next. This shuffling is implemented using the sorting network. The tuples are sorted by a rotated version of the key, rotated by a pseudo-random number, i.e., from an LFSR, thereby shuffling tuples while still keeping tuples of the same key grouped.

4.2 Multi-port Waterfall Cache

An N -port cache is designed to store incoming tuples of recently used keys that would otherwise go to busy banks. As shown by the analysis in the previous section, the cache needs to hold more than N entries. However, a fully associative multi-port cache design would not scale well to a large number of entries. Therefore, the proposed cache is split into multiple (P) pipeline stages, each stage storing N entries, i.e., N ways, as shown in Figure 6. A key can be stored in any stage of the cache, so as expected, a single key may occupy multiple cache entries, i.e., up to P , one per stage. The evictions from one stage are fed to and can be stored or merged with another entry of the key in a subsequent stage, like a waterfall. Hence, the proposed cache is denoted waterfall cache.

A stage contains N entries (ways), $E_1, E_2, \dots, E_N$, each entry storing up to N values of a different key. Supporting up to N parallel accesses with input keys, $k_1, k_2, \dots, k_N$ to a single cache stage requires comparing each input key k_i against all cache entries $E_1, E_2, \dots, E_N$ in the stage, for a total of N^2 parallel comparisons.

On a cache hit, i.e., an input key matches a cached key (entry), the value(s) of the input key are stored next to the already cached values in the entry. In case the total number of values exceeds

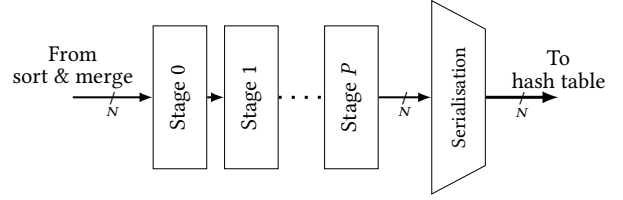


Figure 6: Overview of the waterfall cache. It accepts N parallel requests and is composed of multiple stages. Each stage offers N entries/ways, each entry storing a key and up to N values. Tuples not cached in or evicted from one stage are tried again in the next.

or is equal to the cache entry capacity (N), the N older values of the key are evicted. On a cache miss, the respective valid input key k_i may replace (a) only one specific cache entry E_i , the one with the same index in the stage (i), (b) and only if it has priority over it, i.e., maps to a busier bank, and (c) does not already have N values. The first restriction reduces the hardware complexity of the N -port cache at the cost of cache efficiency, which is alleviated by the randomization of keys described in the previous stage. The second restriction reserves the cache space for keys destined to busy banks in order to merge accesses and alleviate pressure to these banks improving throughput. The priority level of an input key or cached key is determined by finding the queue load of their destination bank after first hashing them using the current mapping. This is performed a cycle prior to the cache access, leaving only the comparison between the bank loads to be in the current cycle for a cache replacement decision, which is in fact performed in parallel to the N^2 comparisons that determine cache hits. The third restriction avoids evicting unnecessarily an entry that could still collect values in favor of one that cannot. Finally, to ensure that hash table accesses spend a limited number of cycles in the cache, each cache entry includes a counter that is reset when the entry is accessed and incremented otherwise; upon reaching an upper counter threshold, the entry is evicted. Figure 7 shows the decision diagram for an incoming tuple entering a cache stage.

The evicted tuples may contain multiple values $\langle k, v_1, v_2, \dots \rangle$. In order to reduce the datapath width, after the final cache stage, evicted tuples are packetized in a multi-flit variable-size format, with a header flit containing key, and size or a single value, and following flits containing values. The up to N cache evictions per cycle are put to one of N lanes, prioritizing less busy lanes, and then multiplexed and sent to the banks through a pipelined link.

Figure 8 shows an example of keys going through the sort, merge and, for simplicity, a single stage cache. Incoming keys 1 to 4 access the cache where k_4 , k_2 , and k_3 are stored, while a fourth cache entry is invalid. Keys 2, 3 and 4 hit in the cache. k_2 causes the number of stored values in the cache entry to reach their limit and so the entry is evicted. k_3 's new values are added to the existing cache entry of k_3 . Finally, incoming key K_1 did not replace the cached entry of K_4 due to lower priority, so it is evicted together with k_2 .

There are several design alternatives for the above multi-port cache. Moreover, for a cache of a few tens of entries, such as the ones built in this work, an implementation using registers and

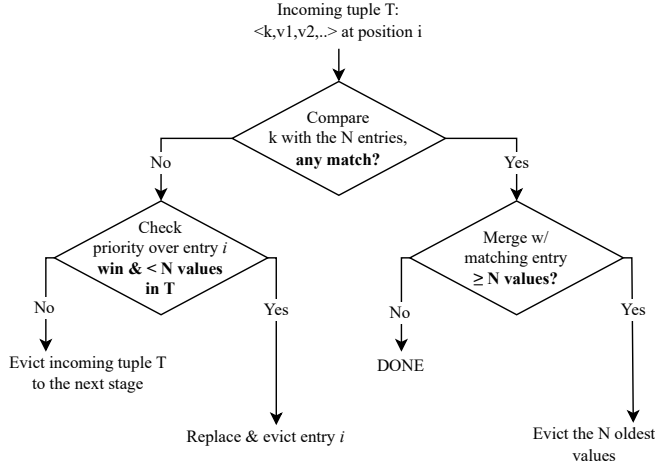


Figure 7: Decision diagram for an incoming tuple entering at position $i = 1, 2, \dots, N$ to a waterfall cache stage.

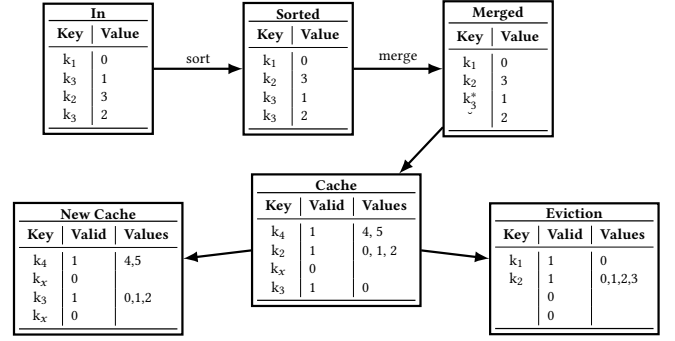
logic is possible; however, larger caches would require an excessive amount of resources, and other options, such as the multi-port SRAM design could be explored [27].

4.3 Hash Table Banks

The evicted tuples are sent to the hash table banks via a pipelined link of N lanes after their keys are hashed using the current address mapping to determine the destination bank. Before the bank, tuples are enqueued to a parallel-in, serial-out queue, in practice implemented with N parallel queues multiplexed to a single output and logic to keep track of the arriving tuples order.

In our implementations, a hash table entry in a bank fits a single key, but associativity could be added, allowing multiple keys per entry to reduce bank conflicts [23]. A bank entry stores the following metadata: key, valid bit, address mapping, status of other mappings, number of values stored locally, flushed last and in DRAM, DRAM head and tail pointers as well as a fixed number of the most recent values of the key, the size of which matches the DRAM access granularity in order to avoid read-modify-write operations in DRAM, which would waste DRAM bandwidth. When the number of values in the bank exhausts the available space, they are flushed to DRAM.

Accessing a bank with an incoming key requires a read access to retrieve the metadata and determine whether it is a hit or miss on the hash table based on a comparison between the incoming key and any valid key stored in the bank entry. A hit will update the hash table entry to include the new incoming values of the key and possibly initiating a data flush operation to the DRAM and/or triggering an aggregation. A miss will need to check whether the newly inserted key already exists in the table with an alternative mapping, as explained in detail next, and also to evict any valid existing entry in the location. The eviction of a valid/active entry causes a flag to rise, indicating a hash collision, and sends the necessary information to software, similarly to previous work [13]. Such an entry is considered valid if it has not expired based on its latest timestamp. In case the evicted valid entry is not placed in the particular hash table entry with the current address mapping, it



*Merging of tuples is done by indicating that the subsequent tuple is of the same key.

Figure 8: Example of incoming tuples going through the sort, merge and (single set) cache stages.

is buffered in a victim cache to cover for any outstanding lookup requests coming from other banks.

4.4 Switching bank address mapping

Load balancing between banks is improved by supporting m alternative address mappings, each using different b bits of the address (the hashed key) to select a bank and hence placing (most) hash table entries to different banks².

The system considers one of these mappings to be the one currently in use, but keys may be already stored in the hash table with another alternative mapping previously used. Accesses that produce a hit, i.e., find an existing entry of the respective key already placed with the current mapping, require a single access to the table. However, accesses that result in a miss need to check whether the key already exists in the system with one of the other $m - 1$ alternative mappings. This generates $m - 1$ lookup requests to the respective banks, which may store the key, and are broadcasted via a separate (ring) network that interconnects the banks. Such lookup requests are filtered at the receiving bank to reduce unnecessary bank accesses, using a small table, which keeps track of the key stored at each valid bank entry. The lookups that pass the filter are then put in the tail of the respective bank queue to be processed in order with other requests in-flight. In case the lookup finds an existing entry of the key, the entry is invalidated, its data (values) are flushed to DRAM, and the head and tail window pointers to DRAM are sent to the new location. These response messages, carrying the pointers back to the new bank, are also filtered at the destination. Most responses are negative, i.e., the lookup was a miss, and such responses do not need to disturb the main read-modify-write loop and are instead marked in a $m - 1$ bits wide table, setting the bit corresponding to what lookup missed. This table can then be read in parallel by the bank controller, while handling a normal access to the same key.

If the response instead was a hit, or the bank is stalling, e.g., due to a flush being blocked by waiting for a lookup, then the lookup response is forwarded to the bank, bypassing the bank queue. The

²For a hash table address to fall on the same bank with two mappings the b bits used by each mapping to select a bank should be identical, which has a probability of $\frac{1}{B}$, for m mappings this becomes $\frac{1}{B^{m-1}}$.

next subsection describes in more detail how flush stalls can be avoided using some extra space in the DRAM.

The current address mapping is changed to an alternative mapping based on the load of the banks. More precisely, any bank queue with load exceeding a set threshold would trigger changing of the address mapping. The threshold is selected to allow a queue to continue accepting input tuples with the worst case rate until the new mapping is in effect, without getting full.

An interesting question arises when more than two mappings are supported, i.e. $m > 2$. Which to pick? One of the $m - 1$ alternative mappings is selected based on statistics kept for the incoming keys that arrived with the current mapping. A saturating counter is maintained per bit of the incoming addresses (hashed version of input keys), which is incremented or decremented when an incoming address has a zero or a one in the specific bit position of the address. Then, bit positions with counter values closer to zero indicate bits with higher entropy, which are preferred when selecting a bank. The absolute values of the counters for the bits used by each mapping for selecting a bank are summed, and the mapping with a lower total score is selected.

4.5 DRAM data store

The SRAM banks of the hash table store metadata for managing the sliding windows and only a subset of the sliding windows, composed of the most recently received values. The complete sliding window of each key is located in DRAM, where values are flushed from the banks in batches that match the DRAM access granularity. The sliding window of a key is stored in DRAM in a fixed-size, statically allocated memory region in a circular buffer and its head and tail pointers are stored in the hash table SRAM bank, similar to previous work [10]. Although the SRAM banks use multiple alternative address mappings, this does not affect the DRAM mapping. Each entry of the hash table, independent of the mapping it uses to access an SRAM bank, has a fixed DRAM location that always uses the same address mapping (address bits order). Consequently, when an old key entry is moved from a previous hash table bank location to the one pointed by the current mapping there is no need of actual data movement between SRAM banks, as explained in the previous subsection, and any values in the old entry are flushed to DRAM.

When a new key is inserted and while waiting for the response of the other $m - 1$ banks to confirm or not whether the input key has any existing entry with an alternative address mapping, more tuples of the same key may arrive. In case the data of these tuples do not fit in the limited storage of the SRAM bank, they need to be flushed to DRAM. However, the tail pointer is not known until the alternative banks respond. For this reason, a small fraction of the DRAM space allocated for storing data of a hash table entry is used as a buffer to temporarily store newly arriving data until the other banks respond. When the response from the other banks arrives, the data in the temporary DRAM buffer is moved to the regular space that stores the key's sliding window. This is the only operation that requires to move data between two different DRAM locations. It is worth noting that a key can have only one other active location in the SRAM banks because upon a new key insertion, alternative locations are checked and moved to the new location of the key.

Another interesting point to note is that, unlike the SRAM banks, the distribution of the incoming keys does not have a significant effect in DRAM performance. Data are aggregated per key first, partially in the SRAM banks, which handle via address remapping and caching any skewed key distribution, and then flushed to DRAM in batches of multiple values, e.g., 64 in our implementation. That allows the values of an equal number of tuples to be aggregated in SRAM before flushing triggering a DRAM access only every $\frac{64}{N}$ cycles and thus alleviating imbalances in the key distribution.

4.6 Compute Stage

Depending on the aggregation function of the query, the incoming values per key can be processed on the fly, gradually as they arrive, e.g., for algebraic (i.e., average) or distributive functions (i.e., minimum, maximum, and sum), or otherwise only when all values have been received, e.g., for holistic functions, such as median. Multiple queries and aggregation functions can be supported by implementing different parallel compute stages. After the computation of aggregation function(s), results are forwarded to output.

The number of incoming tuples per cycle (N) and the number of tuples that trigger the window to advance (WA) determine the frequency of aggregations, thus the frequency of computing a function using the contents of a sliding window, which belongs to a particular key. On average, $\frac{N}{WA}$ aggregations would need to be computed per cycle. However, the peak number of aggregations can be significantly higher. The N incoming tuples may all trigger aggregations at the same time. In addition, multiple (up to N) values evicted together from the cache may cause up to $\frac{N}{WA}$ aggregations. The above call for support of multiple parallel copies of the same compute stage. In our implementation, $\frac{N}{2}$ parallel compute modules for an aggregation function are used. Each bank is able to send a sliding window for aggregation to any of the compute modules. Since the focus of this work is to provide N -port hash table support rather than accommodate very frequent aggregations (small WA), a lot of effort has not been spent on optimizing these compute modules. However, as shown by previous work, caching the most recently used sliding windows can alleviate DRAM pressure for skewed key distributions and small WA [14].

4.7 Discussion

In practice, the Multi Hash Table design may differ from the ideal model considered in the analysis of the previous section. Building an ideal N -port cache that stores all most recent requests to the banks that receive more accesses than their fair share would need to be fully associative, i.e., an N -port Content Addressable Memory (CAM), and it would be expensive even if it is small. Our way of implementing this, by layering a set of very small fully associative caches, slightly limits the amount of usable space as the same key can temporarily take up an entry in multiple stages. For simplicity, the replacement options within a stage are also restricted to a single, particular way for each incoming key, which further limits cache efficiency. In addition, our current implementation prioritises busier banks using only three levels of priority, which is not very accurate compared to using the actual number of elements in a bank queue and therefore may not always prioritise correctly between banks. All the above implementation limitations could be overcome

with a more complex design. More replacement options could be used at the cost of more multiplexers, which would possibly affect the critical path, and exact bank queue load information could be used at the cost of wider comparisons for the replacement decision. Another design aspect that is not ideal is the switching between address mappings, which in practice entails latency and bandwidth overheads. Switching is not instant, and its delay is taken into account in setting the bank queues load thresholds. It is worth noting that there is a bandwidth and latency cost for inter-bank communication. The bandwidth cost of inter-bank communication is minimal as it involves only pointer, rather than data, exchange. The latency cost is tolerated by allowing new hash table entries to store new data in a bank before checking old entries and using temporary buffers in DRAM if this is not enough.

The Multi Hash Table design can be further optimised in various ways. The multi-port cache is expected to limit the operating frequency of the design to at least half of the maximum frequency of the FPGA BlockRAMs. Consequently, the SRAM banks of the design can be used in double the frequency to (virtually) offer double the number of ports, as suggested in previous work [27]. Other parts of the design, such as the inter-bank communication and the cache-to-banks link could also be double-pumped to save resources.

5 Evaluation

5.1 Experimental setup

Multi Hash Table was implemented on the AMD Alveo U280 Data Center Accelerator Card [2] with 8GB HBM2, which provides 32 channels of 460GB/s aggregate throughput. The Alveo card is fed by a 100Gbps Mellanox MCX516A-CDAT, which is connected via a network interface built based on XUP Vitis Network example [41].

Our design supports $N = 8$ incoming tuples per cycle, using $m = 3$ address mappings, $B = 32$ banks of 32K hash table entries in total, and an 80-entry cache of 10 stages with 8 entries per stage. The 32 HBM channels are connected to the banks and to the compute stage(s) using 4 AXI4 interfaces each (8 in total). Each tuple is 4 bytes, more precisely, 3 bytes of key and one byte of value. For comparison, two baseline designs are also implemented offering the same capacity (32k entries). The first one supports $N = 1$ incoming tuples per cycle ($m = 1, B = 1$), which is what previous FPGA-based single-window stream aggregation approaches use [10–14]. The second baseline processes $N = 8$ incoming tuples per cycle, and uses the same number of banks ($B = 32$) but does not have any mechanism to avoid bank conflicts, suffering worst case performance equal to $N = 1$. Besides resource utilization and operating frequency, power estimations were derived from the FPGA EDA tool (Vivado).

Two queries were implemented. The first one comprised of algebraic and distributive functions: “Find the average, minimum, maximum value for each key for the last WS tuples and return the aggregate every WA tuples”. The second one uses a holistic aggregation function: “Find the median value for each key for the last WS tuples and return the aggregate every WA tuples”. Query-1 design supports window sizes from 64 to 1K tuples and the WA varying from 1 up to WS tuples with support for up to 32K concurrently active keys. Query-2 is more complex, and it was possible to reach timing closure when supporting window sizes of up to 256 tuples.

The first query uses 64 parallel compute units, incrementally calculating the three sub queries from 64 values read from DRAM every cycle. The 64 partial results were then reduced to the final result. The version of the system that implements this query has four parallel copies of the compute stage, each with a separate memory interface. For the second query, a median accelerator was implemented as a pipelined sorting network fed with data from four memory interfaces, and the system only has one instance of this compute stage. Both queries support runtime configuration of WS, with that, limiting how much they can be optimized. For most real applications WS and WA should be fixed at compile time.

Most parts of our designs were implemented in the python based hardware definition language Amaranth [1], but some parts, most significantly the compute stages, are implemented using Vitis HLS [3] introducing some inefficiencies.

A number of different key distributions were used to test the robustness of the Multi Hash Table. The test inputs were composed of a repeated sequences of tuples that belong to:

- $i = 1, 2, \dots$, or N distinct keys sent in fixed order;
- distinct keys that map to $j = 1, 2, \dots$, or N banks using each mapping sent in a fixed order; In practice, the hashed value of the keys generates addresses with j distinct values for the b bits of each of the m mappings;
- the above two datasets with 10% and 20% uniform random keys added;
- 32K and 16K distinct keys sent in a fixed order, i.e., equal to the hash table capacity and half that, respectively.
- the linear road benchmark [5], vehicle ID used as key.
- a part of the 2011 Google cluster-usage traces [37], with job ID used as key.

These test inputs were tested in simulation (for better monitoring) running until the system reached a stable state without input queues increasing further. These tests confirmed that the system supported maximum throughput, processing $N = 8$ incoming tuples per cycle as long as WA was large enough for the compute stage to sustain it. In addition, the same test inputs were combined and used in our FPGA prototype experiments to measure the reported performance for difference combinations of WA and WS.

5.2 Implementation results

The implementation results of the evaluated designs are shown in Table 1. Multi Hash Table uses 40-50% of the FPGAs LUTs (spread over 70-80% of available slices) and most of the SRAM resources (90%). The median compute stage is 2.7× larger than the one that supports average, min and max, still compute takes only about 3-12% of the total resources. About a quarter of the resources go to the multi-ported cache component, including sorting and merging, and most of the rest are used by the banks, including the links that interconnect them, their input queues and their interfaces to flush data to the HBM. Both versions of our design operate at 150 MHz supporting $N = 8$ incoming tuples, 256 bits in total, every FPGA cycle, utilizing about 40% of the 100 Gb/s network bandwidth. This translates to a line rate of 1.2 Giga tuples/sec. Finally, the power consumption of the design is estimated to be 46-47 watts.

Table 1: Implementation Results.

Design	N/m/P/W/B/S	Query	Slices [$\times 1000$]	SRAM [Kb]	F max [MHz]	Power [W]
Multi Hash Table	8/3/10/8/32/32k	q1	119 (73.43%)	151776	150	47.5
		q2	127 (78.51%)	142560	150	46.2
Baseline 1	1/1/0/0/1/32k	q1	58 (35.60%)	24264	160	37.8
		q2	60 (37.24%)	21960	160	37.7
Baseline 2	8/1/0/0/32/32k	q1	86 (52.77%)	141980	160	47.8
		q2	94 (58.18%)	132768	160	46.9

Design parameters: # incoming tuples per cycle (N), # addr. mappings (m), # cache pipeline stages & # entries per stage (P,W), # banks (B), hash table size (S).

In comparison, the two baselines have higher clock frequency and operate at 160MHz, because they do not use the cache structure that defines Multi Hash Table critical path. The first baseline ($N = 1$) requires about half of the Multi Hash Table logic despite supporting significantly lower bandwidth because network and HBM interfaces add constant overheads to all designs. However, its SRAM utilization is significantly reduced to $\frac{1}{6}$ versus our design and its power consumption is 20% lower. The second baseline ($N = 8, m = 1$) requires about $\frac{2}{3}$ of the logic and similar amount of SRAM compared to Multi Hash Table, because it does not use the proposed (LUT-based) cache, but offers the same number of banks. Despite fewer resources baseline-2 has similar power cost with the Multi Hash Table because it operates at higher frequency.

It is worth noting that all above designs process 4B tuples that carry 1B values. A Multi Hash Table design processing 8B tuples (4B key, 4B value) would be too large to fit in our FPGA device requiring $2\times$ more logic and $\frac{4}{3}$ of the SRAM resources.

5.3 Performance results

Multi Hash Table was evaluated using the datasets described in the experimental setup and compared to the two baselines. The behavior of the design was first analyzed using RTL simulation and subsequently tested in the FPGA card to measure throughput.

The simulation analysis confirmed that Multi Hash Table could detect busy banks and prioritize the caching of keys going to these banks allowing multiple requests of the same key to be aggregated thereby quickly eliminating the pressure on the respective banks. In cases where input traffic is aimed at overloading multiple banks, the system prioritized caching of their keys in sequence rather than all of them at the same time, allowing the cache to store more keys of the same bank and offload banks faster one by one. The design was also able to detect the address mapping with fewer conflicts and switch to that allowing already entered keys to relocate fast. Even for traffic where each tuple belonged to a different unique key, the inter-bank interconnect was able to support the necessary communication load.

The processing throughput of the Multi Hash Table, i.e., the number of input tuples processed by the system per unit of time, is measured for every query and compared to the two baselines. Figure 9 depicts the Multi Hash Table processing throughput for query-1 and query-2 for different Window Sizes (WS), Window Advance (WA) as well as the throughput of the first baseline ($N = 1$) and the best- and worst-case throughput of the second baseline ($N = 1, m = 1$). For the first query, which implements the simplest compute stage, maximum throughput of 1.2 Gtuples/sec is achieved for all window sizes. This demonstrates in practice the effectiveness

of Multi Hash Table, which is able to sustain the processing of $N = 8$ incoming tuples per cycle for any key distribution. As the WA reduces, aggregations are more frequent and processing throughput gets limited by the bandwidth of the HBM and compute stages, which need to deliver and process, respectively, the contents of a sliding window more often and therefore become the bottleneck of the design. For example, the throughput for queries with WA of 1 and 8 is $5\text{-}25$ and $4\text{-}12\times$ lower than the maximum, respectively. It is worth noting that memory pressure due to frequent aggregations on skewed key distributions can be alleviated by caching most recent sliding windows [14], but this is not implemented in our designs as the focus is to maintain high throughput on the hash table updates. Another interesting observation is that the impact of low WA is more severe for larger WS. This is because larger windows put more pressure both to the HBM and on the compute stages of the system. The second query implements a more complex function, and therefore, the largest supported window size that allowed time closure at the target frequency is smaller (256). The processing throughput of the Query-2 design is similar to Query-1, but slightly lower for small WAs due to a slower compute stage. For large WAs, Query-2 is able to achieve maximum throughput, too.

Compared to the performance of the two baselines Multi Hash Table is slightly less than $8\times$ better ($7.5\times$) than the $N = 1$ baseline, due to its lower frequency. It is equally faster than the worst-case throughput of the second baseline ($N = 8, m = 1$), which performs as fast as $N = 1$ when all accesses go to the same bank because it lacks a mechanism to deal with conflicts. Finally, Multi Hash Table is 6% slower than the best-case performance of the second baseline ($N = 8, m = 1$), i.e., when accesses are evenly distributed to at least $N = 8$ banks, due to its slightly lower frequency. Note that the frequency advantage of the baselines does not yield any throughput advantage for small WAs because in these cases the bottleneck of all designs is in the HBM, which operates at a fixed frequency.

5.4 Comparison with related work

The Multi Hash Table enabled the processing of up to $N = 8$ tuples per cycle increasing stream aggregation throughput eight-fold to 1.2 Gtuples/sec. Current state of the art work on FPGA-based sliding window stream aggregation (SWAG) uses a single port hash tables [13, 14]. They use $4\times$ larger tuples, i.e. 128 bits, and are limited to processing one incoming tuple every two FPGA cycles, at similar frequency supporting 70 Mtuples/sec for the same queries [13, 14]. This is similar to current state of the art GPU-based SWAG [7]. In comparison, the Multi Hash Table is able to offer $17\times$ higher processing throughput versus previous stream aggregation systems.

FPGA and GPU designs proposed for in-memory database queries, share some similarities with our work although they do not tackle the same problem. Stream processing is more challenging than in-memory databases as it requires to process incoming tuples on the fly and each individual incoming tuple triggers a new update to the contents of its entire sliding window. Nevertheless, for queries that require entry updates, FPGA-based [43] and GPU-based [6] in-memory database systems achieve processing throughput of 816 Mtuples/sec and 420 Mtuples/sec, respectively; Multi Hash Table offers $1.5\times$ and $3\times$ higher processing throughput, respectively.

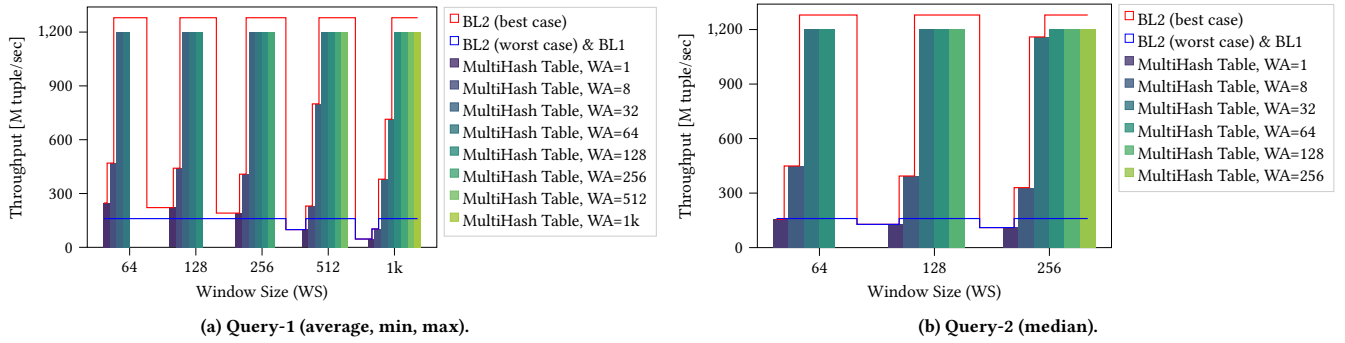


Figure 9: Multi Hash Table ($N=8$, $m=3$) throughput vs. two baselines: BL1 ($N=1$) and BL2 ($N=8$, $m=1$) (worst-case and best-case).

6 Conclusions

Hash tables are important in a wide range of data intensive applications. However, they have difficulties to scale their access throughput as they typically offer a single access port. Previous attempts to increase the number of ports require excessive memory resources, as they have to replicate hash table contents at least as many times as the number of access ports or their performance suffers from bank conflicts and in the worst case is limited to the performance of a single port. This work described a new parallel multi-port hash table design for stream processing, which provides data-independent N -port bandwidth without replicating its contents. Multi Hash Table uses multiple banks and avoids bank conflicts (i) supporting dynamic remapping of the hash table address to redistribute and re-balance accesses among banks, and by (ii) caching accesses to frequently used entries. Dynamic address remapping requires only metadata exchange between the banks, rather than data exchange, because data are flushed to the next level (DRAM), which maintains a fixed address mapping. Multi Hash Table is applied to a reconfigurable single sliding window stream aggregation system demonstrating a $7.5\times$ increase in processing throughput.

Acknowledgments

This work was supported by the Swedish Foundation for Strategic Research (contract number CHI19-0048) under the PRIDE project.

References

- [1] amaranth lang. 2023. amaranth. <https://github.com/amaranth-lang/amaranth>.
- [2] AMD. 2023. Alveo U280 Data Center Accelerator Card. <https://www.xilinx.com/products/boards-and-kits/alveo/u280.html>.
- [3] AMD. 2023. Vitis HLS. <https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html>.
- [4] Henrique CM Andrade, Buğra Gedik, and Deepak S Turaga. 2014. *Fundamentals of stream processing: application design, systems, and analytics*. Cambridge University Press.
- [5] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S Maskey, Esther Ryzkina, Michael Stonebraker, and Richard Tibbetts. 2004. Linear road: a stream data management benchmark. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*. 480–491.
- [6] Saman Ashkiani, Martin Farach-Colton, and John D. Owens. 2018. A Dynamic Hash Table for the GPU. In *IEEE International Parallel and Distributed Processing Symposium, IPDPS*. 419–429.
- [7] Tiziano De Matteis et al. 2019. GASSER: An Auto-Tunable System for General Sliding-Window Streaming Operators on GPUs. *IEEE Access* 7 (2019), 48753–48769.
- [8] Tomáš Fukac, Jiri Matousek, Jan Korenek, and Lukás Kekely. 2021. Increasing Memory Efficiency of Hash-Based Pattern Matching for High-Speed Networks. In *International Conference on Field-Programmable Technology, (FPT)*. 1–9.
- [9] B. Gedik. 2014. Generic windowing support for extensible stream processing systems. *Softw., Pract. Exper.* 44, 9 (2014), 1105–1128.
- [10] Prajith Ramakrishnan Geethakumari, Vincenzo Gulisano, Bo Joel Svensson, Pedro Trancoso, and Ioannis Sourdis. 2017. Single window stream aggregation using reconfigurable hardware. In *2017 International Conference on Field Programmable Technology (ICFPT)*. 112–119. <https://doi.org/10.1109/FPT.2017.8280128>
- [11] Prajith Ramakrishnan Geethakumari, Vincenzo Gulisano, Pedro Trancoso, and Ioannis Sourdis. 2019. Time-SWAD: A Dataflow Engine for Time-Based Single Window Stream Aggregation. In *Int'ernational Conf. on Field-Programmable Technology (FPT)*. IEEE, 72–80.
- [12] Prajith Ramakrishnan Geethakumari and Ioannis Sourdis. 2021. A Specialized Memory Hierarchy for Stream Aggregation. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. 204–210. <https://doi.org/10.1109/FPL53798.2021.00041>
- [13] Prajith Ramakrishnan Geethakumari and Ioannis Sourdis. 2021. StreamZip: Compressed Sliding Windows for Stream Aggregation. In *2021 International Conference on Field-Programmable Technology (ICFPT)*. 1–9. <https://doi.org/10.1109/ICFPT52863.2021.9609952>
- [14] Prajith Ramakrishnan Geethakumari and Ioannis Sourdis. 2023. Stream Aggregation with Compressed Sliding Windows. *ACM Trans. Reconfigurable Technol. Syst. (TRETS)* 16, 3 (2023), 37:1–37:28.
- [15] John R. Gilbert, Steve Reinhardt, and Viral B. Shah. 2006. High-Performance Graph Algorithms from Parallel Sparse Matrices (PARA). In *8th International Conference on Applied Parallel Computing: State of the Art in Scientific Computing*. 260–269.
- [16] Allan Gottlieb, Ralph Grishman, Clyde P. Kruskal, Kevin P. McAuliffe, Larry Rudolph, and Marc Snir. 1983. The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer. *IEEE Trans. Comput.* C-32, 2 (1983), 175–189. <https://doi.org/10.1109/TC.1983.1676201>
- [17] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. 1997. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. *Data Min. Knowl. Dis.*, 1(1) (Jan. 1997), 29–53.
- [18] Vincenzo Gulisano, Zbigniew Jerzak, Roman Katerinenko, Martin Strohhach, and Holger Ziekow. 2017. The DEBS 2017 Grand Challenge. In *ACM Int. Conf. on Distributed Event-based Systems (DEBS)*. 271–273.
- [19] Vincenzo Gulisano, Zbigniew Jerzak, Spyros Voulgaris, and Holger Ziekow. 2016. The DEBS 2016 Grand Challenge. In *ACM DEBS*. ACM, 289–292.
- [20] Vincenzo Gulisano, Yiannis Nikolakopoulos, Ivan Walulya, Marina Papatriantafyllou, and Philippas Tsigas. 2015. Deterministic Real-time Analytics of Geospatial Data Streams Through ScaleGate Objects. In *ACM DEBS*. ACM, 316–317.
- [21] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th Int'l Conf. on Learning Representations, ICLR*.
- [22] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning Both Weights and Connections for Efficient Neural Networks. In *28th International Conference on Neural Information Processing Systems - Volume 1 (NIPS)*. 1135–1143.
- [23] Zsolt István, Gustavo Alonso, Michaela Blott, and Kees Vissers. 2015. A hash table for line-rate data processing. *ACM TRETS* 8, 2 (2015), 13.
- [24] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. 2018. GraFBoost: Using Accelerated Flash Storage for External Graph Analytics. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 411–424.

- [25] A. Kirsch, M. Mitzenmacher, and G. Varghese. 2010. Hash-Based Techniques for High-Speed Packet Processing. In *Algorithms for Next Generation Networks*.
- [26] Donald E. Knuth. 1998. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., USA.
- [27] Charles Eric LaForest and J. Gregory Steffan. 2010. Efficient Multi-Ported Memories for FPGAs. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA)*. 41–50.
- [28] Jin Li, David Maier, Kristin Tufte, Vassilis Papadimos, and Peter A. Tucker. 2005. No pane, no gain: efficient evaluation of sliding-window aggregates over data streams. *ACM SIGMOD* 34, 1 (2005), 39–44.
- [29] Ben Lin, Michael B. Healy, Rustam Miftakhutdinov, Philip G. Emma, and Yale Patt. 2018. Duplicon Cache: Mitigating Off-Chip Memory Bank and Bank Group Conflicts Via Data Duplication. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 285–297. <https://doi.org/10.1109/MICRO.2018.00031>
- [30] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2009. Streams on wires: a query compiler for FPGAs. *VLDB* 2, 1 (2009), 229–240.
- [31] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi. 2010. Cuspars library. In *GPU Technology Conference (GTC)*.
- [32] Michael Offel, Andreas Ley, and Sven Hager. 2023. HashCache: High-Performance State Tracking for Resilient FPGA-Based Packet Processing. In *2023 33rd International Conference on Field-Programmable Logic and Applications (FPL)*. 364–364. <https://doi.org/10.1109/FPL60245.2023.00069>
- [33] Yasin Oge, Masato Yoshimi, Takefumi Miyoshi, Hideyuki Kawashima, Hidetsugu Irie, and Tsutomu Yoshinaga. 2013. An efficient and scalable implementation of sliding-window aggregate operator on FPGA. In *CANDAR*. IEEE, 112–121.
- [34] Rasmus Pagh and Flemming Friche Rodler. 2004. Cuckoo hashing. *Journal of Algorithms* 51, 2 (2004), 122–144.
- [35] Salvatore Pontarelli, Pedro Reviriego, and Juan Antonio Maestro. 2016. Parallel d-Pipeline: A Cuckoo Hashing Implementation for Increased Throughput. *IEEE Trans. Comput.* 65, 1 (2016), 326–331. <https://doi.org/10.1109/TC.2015.2417524>
- [36] M.O. Rabin and V.V. Vazirani. 1989. Maximum Matchings in General Graphs Through Randomization. In *Journal of Algorithms*, Vol. 10. 557–567.
- [37] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. 2011. *Google cluster-usage traces: format + schema*. Technical Report. Google Inc., Mountain View, CA, USA. Revised 2014-11-17 for version 2.1. Posted at <https://github.com/google/cluster-data>.
- [38] André Seznec. 1993. A Case for Two-Way Skewed-Associative Caches. In *20th Annual International Symposium on Computer Architecture (ISCA)*. 169–178.
- [39] I. Sourdis, D. Pnevmatikatos, S. Wong, and S. Vassiliadis. 2005. A reconfigurable perfect-hashing scheme for packet inspection. In *Int'l Conf. on Field Programmable Logic and Applications (FPL)*. 644–647.
- [40] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning Structured Sparsity in Deep Neural Networks. In *Advances in Neural Information Processing Systems*.
- [41] Xilinx. 2023. XUP Vitis Network Example. https://github.com/Xilinx/xup_vitis_network_example.
- [42] Ichitaro Yamazaki and Xiaoye S. Li. 2010. On Techniques to Improve Robustness and Scalability of a Parallel Hybrid Linear Solver. In *Proceedings of the 9th International Conference on High Performance Computing for Computational Science (VECPAR)*. 421–434.
- [43] Yang Yang, Sanmukh R. Kuppannagari, and Viktor K. Prasanna. 2020. A High Throughput Parallel Hash Table Accelerator on HBM-enabled FPGAs. In *2020 International Conference on Field-Programmable Technology (ICFPT)*. 148–153.
- [44] Yang Yang, Sanmukh R. Kuppannagari, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2020. FASTHash: FPGA-Based High Throughput Parallel Hash Table. In *International Conference in High Performance Computing*. 3–22.