Vertical Vectorized Hashing for Faster Group-By Aggregation

Spoorthi Nijalingappa* Bala Gurumurthy* David Broneske† Gunter Saake*
* University of Magdeburg, Magdeburg, Germany, firstname.lastname@ovgu.de
†German Center for Higher Education Research and Science Studies, Hannover, Germany, broneske@dzhw.eu

***Abstract*—`Group-By` is a commonly explored database operator, constantly optimized for better response time. Though multiple strategies are available, its response time heavily depends on the operator's implementation on the underlying device. Hence, in this work, we investigate code optimization for group-by aggregation, specifically in modern CPUs. Recent advancements in modern CPUs support large register sizes with wide SIMD lanes suitable for vectorized execution. Therefore, we utilize these registers to implement a vectorized hash-based aggregation to accelerate the group-by operator. In this work, we investigate vertical vectorization —- insert/probe multiple keys at a time – across various hashing techniques. Our evaluation measures the impact of the load factor, and collisions over the execution in our approach, followed by performance comparison against scalar and horizontal vectorized hashing (a single key is probed in multiple locations at once). Our results show that, although by using SIMD gather and scatter primitives, reaching the theoretical maximum is hard, our approach has up to 10X speed-up in an AVX-512 system (16 vector lanes) compared to scalar execution for various dataset distributions.**

***Index Terms*—SIMD Acceleration, CPU-accelerated DBMS, Vertical vectorization, Hardware-sensitive operators**

I. Introduction

Current generation CPUs come with capabilities such as pipelining, branch prediction, hardware threads, and SIMD registers. In 2008, Intel introduced a new set of high-performance SIMD instructions set[1] – Advanced Vector Extensions (AVX) – with wider SIMD registers for CPUs. Subsequently, several studies have investigated SIMD acceleration for accelerating database operations [1]–[3]. These studies range from exploring SIMD acceleration of complete DBMS operators to exploring individual database operators such as selection, join, aggregation, grouped aggregation, etc. [4]–[6]. However, there are other variants of these operations yet to be realized in SIMD. In this work, we explore one such operator–group-by aggregation.

Group-by aggregation is a critical operator in a query processing pipeline, as it potentially stalls the execution of subsequent operations until all aggregates are generated. Hence, it is important to accelerate the operator using SIMD, as it can improve the overall query execution. Commonly, group-by aggregation is realized using hashing [7]. Hashing is used to partition input keys into buckets, while directly aggregating the payload of the keys. However, hashing is prone to collisions – occurs when two keys are hashed into the same bucket – and is resolved differently based on the underlying hashing technique. Usually, a hashing technique takes time to resolve a collision, which adds to the overall execution time. Hence, we focus on developing appropriate SIMD-accelerated alternatives for these hashing techniques to resolve collisions faster.

Specifically, we consider the three hashing techniques: linear probing [8], 2-choice hashing [9], and cuckoo hashing [10] for processing a group-by input. Here, both linear probing and double hashing resolve their collisions identifying a new location for the incoming key, whereas cuckoo hashing finds a new location for the previously present key in the current slot.

Vectorizing group-by operator poses two main challenges: 1) local conflicts, which occur when identical keys are in the input vector. 2) global conflicts, where two distinct keys are hashed to an identical hash table slot. To resolve these, we develop a five-phase general approach, that works for all hashing techniques. Furthermore, since each individual technique is different, we also develop vectorized implementations for these techniques.

We resolve the local conflicts using scalar linear probing to insert the keys into the SIMD vector. In the case of global conflicts, we use an in-built conflict detection intrinsic to mark the conflicting keys and insert them accordingly. In summary, we examine the efficiency of using AVX-512 for hash-based grouped aggregation and explore the different performance impacts. Overall, we contribute:

- We explore AVX-512 acceleration of three different hashing techniques to support group-by aggregation.
- We compare the performance of our vertical vectorization with scalar and horizontal vectorized [11] execution.

The remaining paper is structured as follows: in Section II, we review the working of the different hashing techniques. Next, in Section III, we extend these hashing techniques with AVX-512 vectorization specifically for group-by aggregation. We evaluate these techniques in Section IV and compare their performance with other approaches. Next, we list the related work in Section V. Finally, we conclude our work in Section VI.

This work was partially funded by the DFG (grant no.: SA 465/51-1 and SA 465/50-1.)

[1]https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html

II. Hashing Techniques — A Primer

Group-by aggregation or grouped aggregation commonly uses hashing to identify groups in the input, these groups are then aggregated. Additionally, certain aggregations – such as sum, count, max, or min – are done *in situ* as soon as the target bucket is identified. In such cases, the bottleneck occurs whenever there is a collision of input. These collisions are resolved by identifying an alternative bucket location for a key based on the hashing technique used. In this section, we give an overview of the common hashing techniques used.

A. Linear Probing [8]

Linear probing is the simplest open addressing technique[2]. It resolves a collision by traversing the hash table linearly until an empty slot is found.

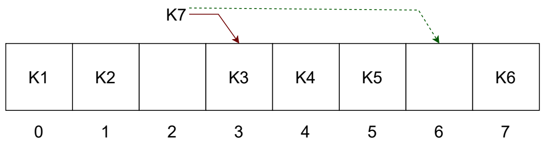


Fig. 1: Example of linear probing

In Figure 1, we show an example of a partially filled linear probing hash table. In this case, K7 is hashed into a slot that is already filled; therefore, the table is probed linearly until the next available empty location is found.

Insertion: For insertion, the input key is first hashed using a hash function, and the corresponding bucket position is determined. If the bucket is free, the key is directly inserted. In case the bucket is full, the hash code is incremented until it finds an available bucket in the hash table.

Probing: Probing also follows the same routine as insertion. Here, the input is hashed and if the key is not present in the bucket, the hash key is incremented until the key is found in the target bucket. In case an empty location is identified while traversing, the key is not available in the hash table. Thus, linear probing is a simple but penalty-prone hashing technique. To overcome the penalty of probing the hash table, an alternative strategy is followed in 2-choice hashing.

B. 2-Choice Hashing [9]

2-choice hashing utilizes randomization in the "power of 2 choices" [12]. Here, instead of identifying a single target slot, we get two alternative slots using two different hash functions. The key is then placed in the favorable slot out of the two.

Insertion: Insertion in 2-choice hashing is similar as liner probing. However, we search for a target slot by alternating between both the target slots from the hash functions. For example, in Figure 2, the key K8 gets two slots: 3 and 6. As slot 3 is occupied, the key is inserted in Slot 6 (green dotted arrow). Such insertion reduces the overall insertion time considerably compared to linear probing.

[2]Instead of chaining keys on the same bucket, an alternative bucket is found within the hash table

Probing: Again, similar to linear probing, the search for an existing key also probes from the target slot(s). Here, the slots are alternatively probed until we identify the key or an empty location from both target locations.

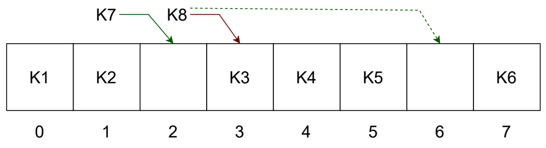


Fig. 2: Example of 2-choice hashing

C. Cuckoo Hashing [10]

The final hashing technique we consider for AVX-512 acceleration is cuckoo hashing[3], which uses several hash tables and hash functions. Unlike previous techniques, in case of a collision, the technique inserts the current key into its target location while evicting the key currently present. The evicted key is stored in an alternative location in another hash table.

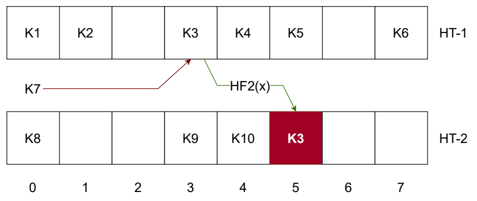


Fig. 3: Example of cuckoo hashing

Insertion: As mentioned, collision results in an eviction of the value currently in the slot. For example, in Figure 3, we see the key K7 is given the location 3 in hash table HT-1, where K3 is already present. Here K3 is evicted, replacing it with K7 in the table, while K3 is stored in the alternative table HT2 (marked in red).

Probing: Though insertion is complicated, probing is straightforward. Probing has constant lookup time (*O(1)*). The key is present in one of the slots given by the hash functions of the two tables.

Insertion Cycle: One drawback of the technique is the possibility of insertion cycles. This occurs when keys are swapped across tables, forming a closed eviction loop. In such cases, the tables are erased, and the keys are re-hashed with a different setup.

So far, we have seen the different hashing techniques used in computing the group-by aggregates. Further, we see that the bottleneck is in searching for an alternative location in the table. Hence, optimizing the hash probe leads to better performance. In the next section, we propose such an optimization using SIMD for a faster probe.

III. Vertical Vectorized Hashing

Vertical vectorization improves execution time by inserting multiple input keys at a time. However, vertical vectorization also comes up with two significant challenges: 1) duplicates

[3]The name for this hashing technique comes from the behavior of the cuckoo bird, where the hatchling pushes other eggs out of the nest as soon as it hatches

in the input vector, and 2) identical slots for two keys in the input vector. Additionally, we need a tailor-made vectorized execution flow depending on the hashing technique in hand. In this section, we show the ways to handle the above-mentioned challenges and develop a general workflow for vertical vectorization. Later in the section, we show the SIMD vectorization of different hashing techniques.

A. General Workflow - Reducing Conflict on Key Insertion

As described in the previous sections, hashing techniques resolve conflicts that arise when two unique keys are inserted in a single location. In addition to this, a vectorized hashing technique must also resolve the collision that arises from inserting two duplicates of a single key. Such collisions are possible as we aim to insert a vector of input in an instant. To overcome these collisions across different hashing techniques, we develop a general workflow that can be extended with any hashing technique.

Foremost, we must resolve the conflicts from duplicates in the input before filling the input vector. To this end, we use scalar linear probing, where duplicates are pre-aggregated within the input vector itself. Next, for resolving collision from two different keys, we insert one of the keys while marking the other as collision. The marked keys are retained in the input vector and are inserted in the following iterations. We combine these two workarounds into a generic execution flow, which is itself split into five phases: load, hash, lookup, store, and carry. The overall execution flow across the five phases is given in Figure 4. Each of the individual phases is detailed below.

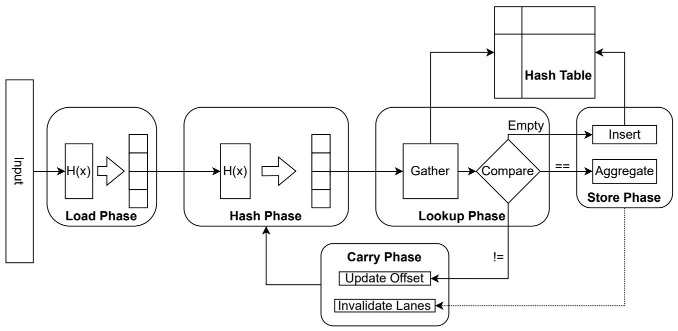


Fig. 4: General workflow

Load Phase: The Load phase is common for all hashing techniques. It is responsible for handling collisions from duplicate keys. As mentioned earlier, we use simple linear probing to populate the input vector. The input key is hashed (we use murmur3 hashing function) with vector width as the hash table size. Whenever a duplicate is encountered, we simply update its partial aggregate. As soon as the whole input is populated, we move to the hash phase.

Hash phase: In the hash phase, we execute a vectorized hash function over the input vector to identify the keys' corresponding buckets. Once the slots are identified, we do a lookup to see if the keys are already present. Depending on the hashing technique considered, there can be more than one hash function present in this phase.

Lookup phase: In the lookup phase, we compare the input keys with the ones in their corresponding slots. Again, depending on the hashing technique considered, we will have to lookup and gather keys from more than one hash table slot. Once the keys are gathered, there are three possible outcomes: 1) the target slot can be empty, in which case the key can be inserted here, 2) the input key is already present leading to updating the aggregate, and 3) a different key is present resulting in a collision. Based on the outcome of this phase, we go to the store/carry phase.

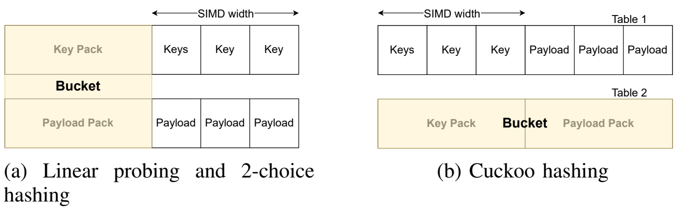
Store phase: This phase is called when their keys are ready to be inserted. In the store phase, we insert keys in empty slots as well as aggregate the ones whose keys are already present.

Carry phase: In this phase, alternative locations are identified for conflicting keys. Except for cuckoo hashing, the other techniques identify alternative slot IDs. For cuckoo hashing, we swap the input keys with the ones present in the slot.

We go through these phases until all inputs are inserted into the hash table. Now, we explain in detail the vectorized execution of the hashing techniques.

B. Table Structure

For better SIMD probing and cache locality, we first modify the hash table structure for the hashing techniques (based on existing work [11]). In the case of linear probing and 2-choice hashing, the table is searched sequentially. Hence, it benefits the techniques to keep the hashed keys together, enabling faster probing. So, we devise the table to be a Structure of Array (SoA), with one array for the keys and another for payloads (cf. Figure 5-a).



(a) Linear probing and 2-choice hashing

(b) Cuckoo hashing

Fig. 5: Hash table structure

Next, cuckoo hashing probes a key across multiple hash tables. Depending on the availability of the key, its corresponding payload is also updated. Hence, in this case, we devise the hash table to be an Array of Structures (AoS) with keys and payloads packed together (cf. Figure 5-b). This way, when swapping keys, we can easily access the corresponding payload.

C. *Vectorized Linear Probing*

We incorporate AVX-512 to enable faster comparison of input vector keys with keys in the target slots. The overall execution of our AVX-512 linear probing is given in Figure 6. The execution starts with hashing the input vector (with only distinct keys) to identify their target slots. Next, these slot values within the values are gathered using AVX's gather function. We compare these gathered values, once with the zero vector and once with the input vector. With the former, we can identify empty locations, and with the latter, whether keys are already present in the hash table.

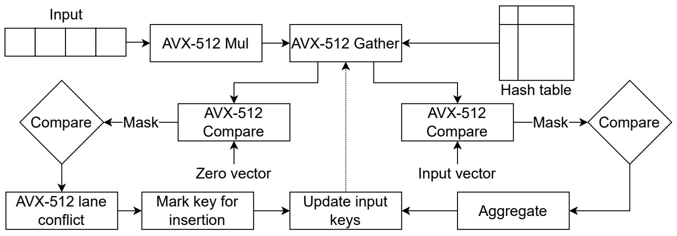


Fig. 6: AVX-512-accelerated linear probing

Key insert: The keys with zero in their hash table slots are then compared for any lane conflict. This checks for conflict in the target slots for two independent keys. In case of conflict, we mark one of the keys to be inserted while the other is marked as a collision. Based on this, we insert the keys and update the input vector. The slots for the remaining keys are updated, restarting the process.

Payload update: The keys that are already present in the table are simply aggregated of their payload values. Once the payload is updated, the corresponding input keys are removed from the input vector. We repeat the execution with new slots for the remaining keys until all the keys in the vector are updated.

D. *2-Choice Hashing*

Since 2-choice hashing is an extension to linear probing, we have the same execution flow as in Figure 6. However, in addition to the single hash function with AVX gather, we execute it twice – once per hash function. The remaining execution flow is the same as above for insertion and aggregation.

E. *Cuckoo Hashing*

Unlike the previous techniques, cuckoo hashing evicts keys already present in the hash table. In the case of vectorized execution, the insertion and probing follow the same as the previous approach. However, the conflicting keys are forwarded to the insertion loop.

Payload update: The aggregation of keys is the same as the one from linear probing. In this case, we compare the gathered keys with our input keys from HT-1 and update the ones already present in them. Next, we remove the ones

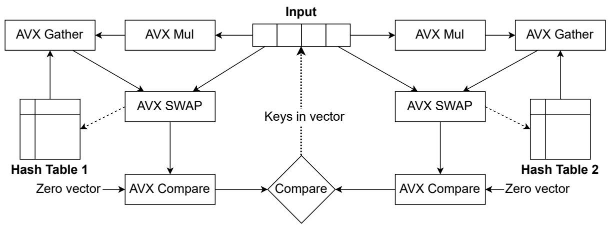


Fig. 7: AVX-512 accelerated cuckoo hashing—Insert loop

already inserted and compare the ones remaining with the keys gathered from HT-2.

Key insert: Once the keys' payload is updated, we enter the insertion cycle. The overall flow of the insertion loop is given in Figure 7. As shown, the keys are hashed and the corresponding keys are gathered. These keys are swapped with input and compared with a zero vector. In case the values after the swap are empty (mask is returned as 65535), we stop the execution. Else, we switch to the next hash table with these keys as new input. We repeat the insertion by alternating the hash tables until all the keys are inserted.

IV. Evaluation

In this section, we evaluate the hashing techniques to understand the impact of vectorization. Using the optimal load factor for the different hashing techniques, we measure the performance of vectorization across various data distributions and compare it against the scalar and horizontal vectorized [11] execution.

Experimental Setup

Our experiments are run on an Intel® Xeon® Gold 5220R CPU @ 2.20GHz running Ubuntu LTS 18.04. Our hashing techniques are implemented in C++ [4] with AVX-512 SIMD extensions. It is compiled using GCC version 10.1 with necessary optimization flags (-O3 -mcmodel=medium -mavx512f -mavx512cd -mavx512bw). We use *Mumur3* as the main hash function across all hashing techniques.

We consider integer input arrays with sizes ranging from 5 million to 50 million, incremented in steps of 5 million. All our experiments are run for 20 iterations and their execution time are averaged. Furthermore, we also experiment with three data distributions: Dense unique random, sequential, and uniform random to study the impact of data ordering. These are generated based on techniques mentioned in [13]. Finally, even though we use count as our aggregate function, other aggregation functions like min, max, and sum will also have similar performance impacts.

A. *Impact of Load Factor*

Load factor[5] is a critical parameter of the hash table that heavily influences the number of collisions. The parameter

[4]Complete code is available in https://github.com/spoorthin/Vertical-Vectoriztion-of-GroupBy-Aggregation-Hashing

[5]the ratio of keys to the number of hash table slots.

is critical in fine-tuning the trade-off between time & space utilized while hashing. Hence, we experiment with different load factors to identify one value where time and space complexity for hashing technique is at a minimum. In this experiment, we insert a uniform random dataset with 50M keys, while varying the hash table size according to the given load factor. The results are plotted in Figure 8.

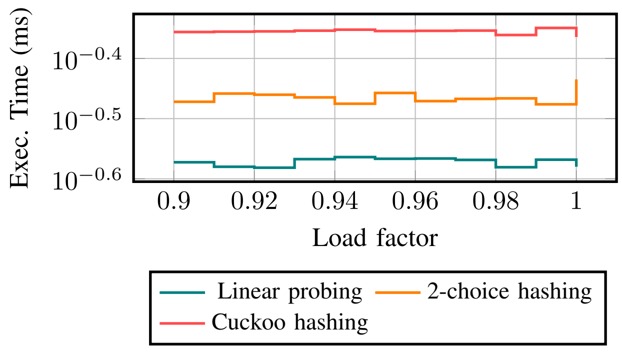


Fig. 8: Impact of load factor

As we see in the figure, the load factor has a varying performance impact and execution profile with the underlying hashing technique. Linear probing's execution time increases with the load factor, dropping to around 0.98. In the case of 2-choice hashing, we see that the optimal load factor is 0.97 for 2-choice hashing. Finally, cuckoo hashing has an optimal load factor of 0.98. Overall, we see that even with quite a high load factor, we get competing performance with all hash tables. We believe such behavior is due to optimal caching. A collision in the 1.0 load factor leads to a lot of random memory accesses, resulting in many cache misses. However, around 0.97 or 0.98 load factors, the likelihood of accessing the target slot is comparatively better. Additionally, pre-fetching also helps that the target slot is already present in the cache lines. Hence, we use these load factor values to explore the performance of hashing techniques with varying data distributions.

B. Impact of Data Distribution

In this section, we compare the performance of our vertical vectorized execution with horizontal & scalar execution with varying data distributions and data sizes. Depending on the data size and the optimal load factor for the hashing techniques (identified from the above section), we create the hash tables. The comparison results for different hashing techniques are given in Figure 9. As expected, the execution time grows linearly with more data, except for the unique random distribution where it is nearly constant. Such behavior across all hashing techniques is due to the compatibility of the randomization function and our hashing function, resulting in fewer collisions. This distribution simulates the best-case scenario, with sequential distribution representing the worst-case, while uniform random represents the scenario in between. Also, the results show horizontal vectorization is on average 3x faster than scalar execution. This can be attributed to only one key being inserted at any given instant. Below is the discussion on the behavior of individual hashing techniques.

Linear probing: Linear probing is 13x faster than scalar and horizontal execution for dense unique and random distribution, and 9x faster for sequential. With dense unique, being the best-case distribution, the speed-up nearly matches the size of the vector width. However, we cannot reach peak performance due to two pitfalls: 1) scalar input load incurs overhead and 2) the data access time for gather & scatter functions.

2-choice hashing: Similar to the case of linear probing, 2-choice hashing also has a similar performance profile across different distributions. This is reflected in the speed-up gain across the distributions. We see an increase in speed-up (up to 10x) for sequential distribution, which is due to the exploitation of alternative slots for each key in the hash table.

Cuckoo hashing: Cuckoo hashing is the fastest among the chosen ones across all execution modes. However, the speed-up gain is sub-optimal when compared with the other two. Such behavior is due to the constant swap and replace of values within the hash table. As mentioned in linear probing, the gather & scatter functions increase execution time with each memory access, and hence they cause the decline in performance. Still, we see up to 10x performance gain compared to scalar and nearly 8x speed-up compared to horizontal vectorized execution. Overall, we see improvement in speed using vertical vectorization. We reach an average performance of more than 8x across the techniques than other counterparts.

Analyzing speed-up: The previous section showed varying speed-ups for our vectorized approach. Here, we detail the rationale behind these speed-up ranges. The box plot in Figure 10 details the speed-up across the data distributions for different hashing techniques. The dotted line in the figure marks the maximum speed-up of 16x, as we can insert 16 keys at a time.

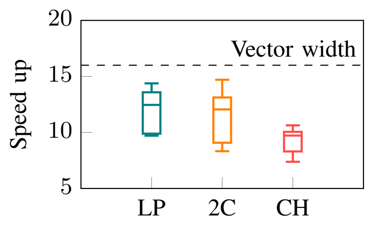


Fig. 10: Speed-up across different hashing techniques

As we see, the maximum gain across all hashing techniques is around 15x. They don't reach the theoretical maximum, mainly due to the penalty of loading the input vector. Furthermore, each gather and scatter operation also incurs latency (of around 30 clock cycles[6]).

When investigating individual hashing techniques, we see that linear probing and 2-choice hashing have high variability of speed-up. This shows that they are influenced by the underlying input distribution. However, cuckoo hashing has less impact on data distribution. However, the speed-up gain is sub-optimal. Since cuckoo hashing suffers from multiple swaps, it has poor speed-up gain compared to other techniques.

[6]https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#techs=AVX_512&ig_expand=4005

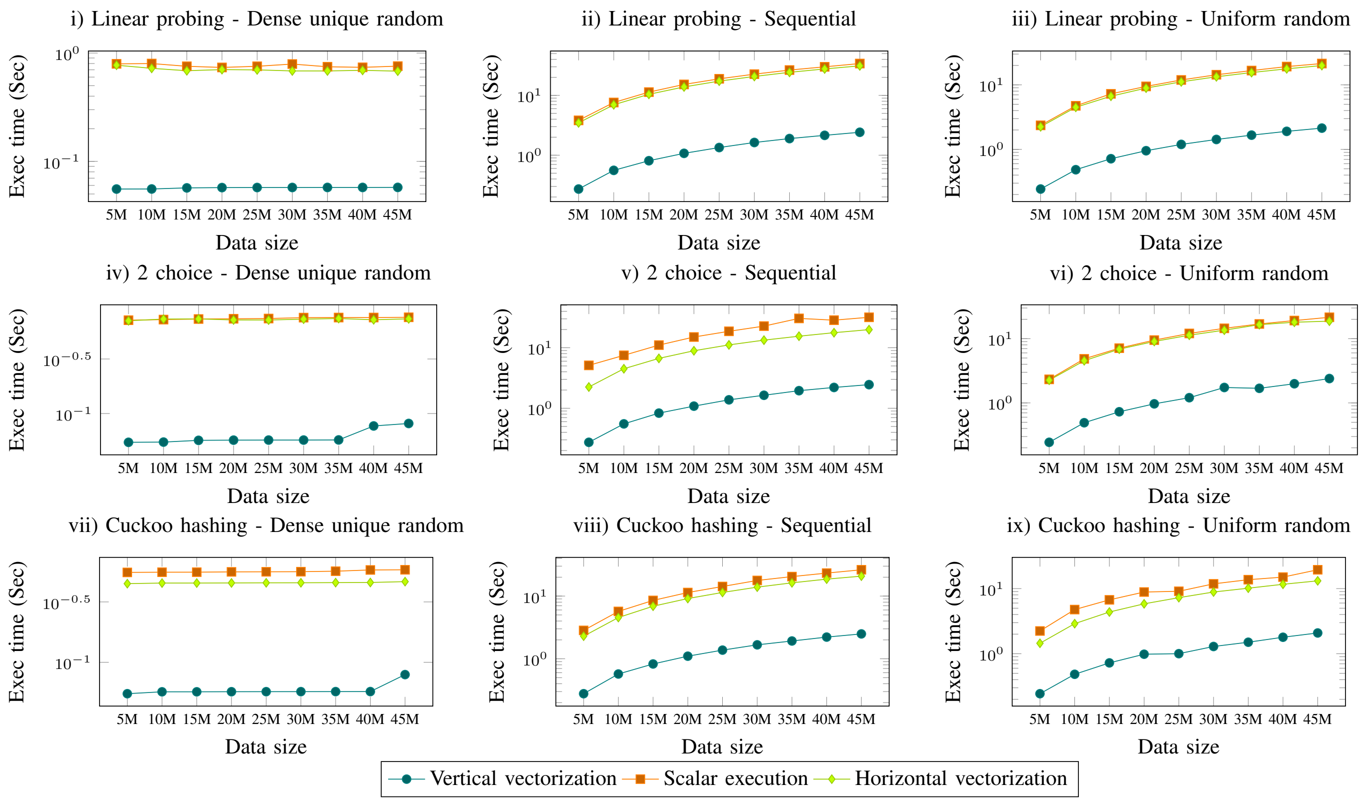


Fig. 9: Performance of SIMD accelerated hashing across data distributions

V. Related work

In this section, we review work with SIMD for aggregation, hashing techniques, and hardware-related optimizations for SIMD.

In [4], the authors have implemented vertical vectorization for linear probing hashing along with other DB operators. Unlike their software-based collision detection statement, we use Intel's instruction set Conflict Detection (AVX-512CD) allowing the vectorization of loops with possible address conflicts. Unlike their work, we have an additional load phase where we pre-aggregate the input keys to avoid conflict from equal keys.

A similar vertical vectorization of linear probing with a new AVX-512 instruction set has been attempted in [14]. Unlike this work, we use a fixed vector lane to store the duplicate keys and process them in a single iteration saving more time. Jiang et al. [15] developed a hash-based grouped aggregation that addresses SIMD vector conflicts. However, they focus on chained hashing whereas we focus on open-addressing techniques. Such similar realizations are also present in [16], [17].

Behrens et al. [18] use OpenCL for data-parallel hashing. They also split their execution into multiple phases. Unlike their work, we make our approach generic to fit multiple hashing techniques. Other than these, SIMD has been used for other database operations as well. Lang et al. [6] explore SIMD on complete query pipelines. Zhou et al. [19] accelerated operations like scans, index search, and nested loop joins. [20] proposes a SIMD sorting called Aligned-Access sort (AA-sort). Our implementation can work with these operations in a query pipeline enabling faster query execution.

VI. Conclusion

Vectorization in modern CPUs increases the performance of existing applications many-fold. Hence, it is a suitable code-optimization strategy for developing efficient database operators. In this work, we focus on developing an efficient hash-based group-by operator. We architect the operator using a five-phase workflow, which can support various hashing techniques. We further use this workflow to implement three AVX-512-accelerated hashing techniques. Our experiments show that vectorization enables speed-ups between 10x and 13x the scalar execution depending on the underlying data distribution. Our experiments also show that the techniques suffer penalties in populating the input vector as well as while accessing multiple memory locations in an instant. As future work, other hashing techniques like hopscotch hashing can be extended. Furthermore, we comprehensive study of various vectorized hashing techniques could be also studied to understand the overall impact of vectorization.

Acknowledgment

This work was partially funded by the DFG (grant no.: SA 465/51-2).

References

[1] K. A. Ross, “Efficient hash probes on modern processors,” in *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 2006, pp. 1297–1301.

[2] D. Broneske, S. Breß, M. Heimel, and G. Saake, “Toward hardware-sensitive database operations.” in *EDBT*, 2014, pp. 229–234.

[3] L.-C. Schulz, D. Broneske, and G. Saake, “An eight-dimensional systematic evaluation of optimized search algorithms on modern processors,” *Proc. VLDB Endow.*, vol. 11, no. 11, p. 1550–1562, jul 2018. [Online]. Available: https://doi.org/10.14778/3236187.3236205

[4] O. Polychroniou, A. Raghavan, and K. A. Ross, “Rethinking SIMD vectorization for in-memory databases,” in *Proceedings of the ACM SIGMOD*. ACM, 2015, pp. 1493–1508.

[5] D. Broneske, A. Meister, and G. Saake, “Hardware-sensitive scan operator variants for compiled selection pipelines,” in *Proceedings of BTW*, 2017, pp. 403–412.

[6] H. Lang, L. Passing, A. Kipf, P. A. Boncz, T. Neumann, and A. Kemper, “Make the most out of your SIMD investments: counter control flow divergence in compiled query pipelines,” *Proceedings of VLDB*, pp. 757–774, 2020.

[7] A. Sharma and H. Zeller, “Hash-based database grouping system and method,” in *Google Patents*, April 1996.

[8] D. E. Knuth, “Linear probing and graphs,” *Algorithmica*, pp. 561–568, 1998.

[9] Y. Azar, A. Z. Broder, A. R. Karlin, and E. Upfal, “Balanced allocations,” in *Proceedings of ACM STOC*, 1994, pp. 593–602.

[10] R. Pagh and F. F. Rodler, “Cuckoo hashing,” *Journal of Algorithms*, pp. 122–144, 2004.

[11] B. Gurumurthy, D. Broneske, M. Pinnecke, G. Campero, and G. Saake, “SIMD vectorized hashing for grouped aggregation,” in *Proceedings of ADBIS*. Springer, 2018, pp. 113–126.

[12] A. Georgakopoulos, J. Haslegrave, T. Sauerwald, and J. Sylvester, “The power of two choices for random walks,” *CoRR*, 2019.

[13] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger, “Quickly generating billion-record synthetic databases,” in *Proceedings of ACM SIGMOD*, 1994, pp. 243–252.

[14] J. Pietrzyk, A. Ungethüm, D. Habich, and W. Lehner, “Fighting the duplicates in hashing: Conflict detection-aware vectorization of linear probing,” in *Proceedings of BTW*, 2019, pp. 35–53.

[15] P. Jiang and G. Agrawal, “Efficient SIMD and MIMD parallelization of hash-based aggregation by conflict mitigation,” in *Proceedings of ICS*, 2017, pp. 24:1–24:11.

[16] O. Polychroniou and K. A. Ross, “Vip: A simd vectorized analytical query engine,” *The VLDB Journal*, vol. 29, no. 6, pp. 1243–1261, 2020.

[17] Z. Fang, B. Zheng, and C. Weng, “Interleaved multi-vectorizing,” *Proceedings of the VLDB Endowment*, vol. 13, no. 3, pp. 226–238, 2019.

[18] T. Behrens, V. Rosenfeld, J. Traub, S. Breß, and V. Markl, “Efficient SIMD vectorization for hashing in opencl,” in *Proceedings of EDBT*, 2018, pp. 489–492.

[19] J. Zhou and K. A. Ross, “Implementing database operations using SIMD instructions,” in *Proceedings of ACM SIGMOD*, 2002, pp. 145–156.

[20] H. Inoue, T. Moriyama, H. Komatsu, and T. Nakatani, “AA-sort: A new parallel sorting algorithm for multi-core SIMD processors,” in *Proceesings of PACT*, 2007, pp. 189–198.