SIMD Vectorized Hashing for Grouped Aggregation

Bala Gurumurthy, David Broneske, Marcus Pinnecke, Gabriel Campero, and Gunter Saake

Otto-von-Guericke-Universität,
Magdeburg, Germany
firstname.lastname@ovgu.de

Abstract. Grouped aggregation is a commonly used analytical function. The common implementation of the function using hashing techniques suffers lower throughput rate due to the collision of the insert keys in the hashing techniques. During collision, the underlying technique searches for an alternative location to insert keys. Searching an alternative location increases the processing time for an individual key thereby degrading the overall throughput. In this work, we use Single Instruction Multiple Data (SIMD) vectorization to search multiple slots at an instant followed by direct aggregation of results. We provide our experimental results of our vectorized grouped aggregation with various open-addressing hashing techniques using several dataset distributions and our inferences on them. Among our findings, we observe different impacts of vectorization on these techniques. Namely, linear probing and two-choice hashing improve their performance with vectorization, whereas cuckoo and hopscotch hashing show a negative impact. Overall, we provide in this work a basic structure of a dedicated SIMD accelerated grouped aggregation framework that can be adapted with different hashing techniques.

Keywords: SIMD · hashing techniques · hash based grouping · grouped aggregation · direct aggregation · open addressing

1 Introduction

Many analytical processing queries (e.g., OLAP) are directly related to the efficiency of their underlying functions. Some of these internal functions are time-consuming and even require the complete dataset to be processed before producing the results. One of such compute-intensive internal functions is a grouped aggregation function that affects the overall throughput of the user query. Hence, it is evident that improving the throughput of grouped aggregation processing in-turn improves the overall query throughput.

A grouped aggregation function is commonly implemented using hashing techniques. The throughput of a grouped aggregation function is affected by the collision of keys in the underlying hashing techniques. A collision occurs when

a given key is hashed to a slot that is preoccupied with another key. In this case, the collision is resolved by finding an alternative location for the colliding key. Searching for an alternative location requires more time and affects the throughput of the whole operation. Hence, improving probe time in a hashing technique improves the overall execution time.

Since real-time systems process large volumes of data, Single Instruction Multiple Data (SIMD) is the commonly sought out parallelization strategy to perform these operations. Previous work has shown how to use SIMD to accelerate different database processing operations [7, 3, 11]. Ross et al., detail a strategy of using SIMD acceleration for probing multiple slots in cuckoo hashing [14]. Furthermore, Broneske et al. have shown that execution of hand-written database operations that are code optimized for underlying hardware can be faster than the query plan given by a query optimizer [1, 2]. Using these insights, we explore the advantages of hand-written grouped-aggregation using SIMD acceleration over various hashing techniques. In addition to it, we also use the code optimization strategy of direct aggregation for improved throughput (explained in Section 3).

We discuss a SIMD-based probing called horizontal vectorization, where multiple slots are searched for a single key at an instant using the instructions available in the SIMD instruction set [10]. We adopt this technique to different open-addressing hashing techniques available. This vectorized probing is enhanced with direct aggregation for increased performance. From our evaluation results, we observe that these optimizations provide notable speedups of up to Nx over the scalar mechanism, where N is the vector length. Our core contributions in the paper are:

- We explore SIMD-based code optimization for different hashing techniques (Section 3).
- We evaluate the performance of computing grouped aggregation using these SIMD accelerated techniques (Section 5.2).
- Finally, we report our observations on the results and discuss the impact of SIMD for the different hashing techniques (Figure 10).

From our evaluation, we found out that the multiple parameters involved in hashing techniques also affect their efficiency. Hence, SIMD accelerated grouped aggregation could be further improved by tuning these underlying hashing technique's related parameters. Also, based on our results, we show that the best hashing algorithm for a given data distribution can be selected for reduced latency.

2 Related Work

In this chapter, we compare our work with others that are similar in studying the aggregation operation in databases. The approaches are selected based on the use of SIMD and other hardware related optimizations over grouped aggregation functions.

In this work, we do not consider quadratic probing and double hashing. We exclude them due to their poor locality of keys.

Jiang et al. use a modified bucket chaining hashing technique to group data and aggregate them on the fly during insertion [7]. In order to eliminate conflicts arising due to SIMD parallelization with insertion and to accommodate values within main memory, they have proposed to add a distinctive offset for each of the SIMD lanes and manipulate data separately. This approach is also extended for MIMD execution. Broneske et al. use SIMD to accelerate selection operations [1]. The authors argue that SIMD acceleration influences the operation of aggregation operations. In our approach, we have gained similar improvements in SIMD acceleration of grouping with aggregation operation especially for linear probing and two choice hashing. Further, various SIMD-based aggregation operations are explained in the paper by Zhou and Ross [15]. They detail the performance impact of SIMD on different aggregation operations.

Finally, Richter et al. provided an extensive analysis of different hashing techniques [13]. These are some of the works done on hashing and grouped aggregation techniques. In the next section, we discuss different hashing techniques followed by details on incorporating SIMD on the them.

3 Hash-Based Grouped Aggregation

Traditional grouped aggregation using hashing techniques is computed in two phases. In the first phase, all the keys are hashed and segregated into their own buckets. In the second phase, each of the keys within the buckets is aggregated providing the final result. Since it is time-consuming to perform two passes on the same data, an alternative single pass variant - *direct aggregation* is used to improve the throughput.

Direct aggregation initializes the aggregate value for a key during its insertion. During subsequent insertions of the same key, its corresponding aggregate value is updated. To further improve the throughput of the single pass algorithm, SIMD vectorization can be used.

In this section, we describe the different hashing techniques used for computing grouped aggregation followed by our strategy to incorporate SIMD vectorization on these techniques.

3.1 Outline of Hashing Techniques

The major bottleneck of any hashing technique is the collision of keys, where two keys are hashed into the same location. Each of the hashing techniques has their own ways to resolve the collision of keys.

We use the open-addressing hashing techniques: linear probing, two-choice hashing, hopscotch hashing and cuckoo hashing for computing grouped aggregation. These techniques have a linear hash-table structure with a known boundary which is suitable for vectorization. In the following sections, we detail the collision resolution mechanism of these techniques and the ways to perform grouped aggregation on them.

3.2 Cuckoo Hashing

Cuckoo hashing resolves collision by using multiple hash tables [9]. These tables resolve collision by swapping collided keys. The collided key, during collision, replaces the key within the insert table. The replaced key is then hashed into the next hash table. The keys are swapped until the final table. In case of collision in final table, the replaced key is routed back to the first table and swapping continues. Since number of swaps is in-determinant, a user-defined threshold, *swap number* cuts-off, swapping and re-hash the whole keys using different hashing functions for the tables and increasing the hash table size. Hence, cuckoo hashing has near direct probing by trading-off increased insertion time. For grouped aggregation computation, first the given key is probed in all the locations in the hash tables. If it is found, the corresponding payload is updated. Else, the key along with the initial payload are inserted. Both the key and payload are swapped among the tables for accommodating the new key.

3.3 Linear Probing

Linear probing, as the name suggests, searches the hash table for a desired location using sequential scanning [4]. It searches the hash table to find the nearest empty slot for insertion. Similarly, it scans the table linearly for the probe value. For grouped aggregation, we search the hash table until either the probe key or an empty location is found. If an empty slot is encountered first, then aggregate resultant is initialized. Else if the key is found, the aggregate is updated.

3.4 Two-Choice Hashing

Two-choice hashing, a variant of linear probing, has two different hash functions for a single hash table [12]. The two hash functions provide for a given key, two alternative positions. Having alternative positions increase the chance of finding a slot with lesser probe time. If both of the slots are occupied, the hash table is probed linearly from both the slots until an empty slot is encountered. Grouped aggregation is computed similar to linear probing. However, instead of a single probe, the hash table is probed from the two start locations.

3.5 Hopscotch Hashing

Hopscotch hashing has a user-defined parameter - neighborhood - that guarantees any given key will be available within the neighborhood range from the originally hashed location [6]. If the key is not available within the neighborhood, it has to be inserted. For insertion of a key, the hash table is searched for an empty location and the existing keys in the table are swapped until an empty slot is available within the neighborhood. Finally, the key is inserted in the slot. Thus, hopscotch hashing trades less search space to prolonged insertion.

4 SIMD Vectorization of Hashing Techniques

SIMD performs a single instruction on multiple data in an instant. Modern CPUs have extended register sizes of up to 128 bits, possible for accommodating four packeted integers. Using these larger registers, a single operation is applied over multiple data at a time. Though it might seem trivial that K packed values provide a speed-up of K in SIMD, incorporating SIMD in hashing techniques has its complexities due to the collision of keys.

Specifically, a hashing technique has delayed time due to the search of the desired location either for inserting a new key or searching an existing key. We address this issue by incorporating SIMD for searching multiple slots at a time thereby improving the throughput. In the next sections, we detail the SIMD accelerated probing on the above discussed hashing techniques.

4.1 Table Structure

Probing requires multiple slots of a given hash table to be readily available. Hence, a right table structure improves the efficiency of the hashing technique. Since cuckoo hashing has multiple hash tables, we use the table structure described in Figure 2. We store a packed set of keys followed by a packed set of payloads for each bucket. As multiple swaps are required, packing keys and payloads improves the efficiency by loading both into the memory for easy swapping among tables.

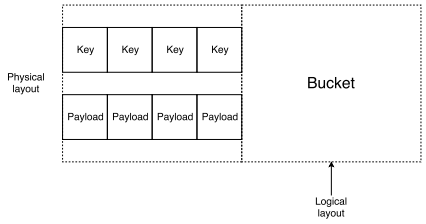

Fig. 1: SoA table structure

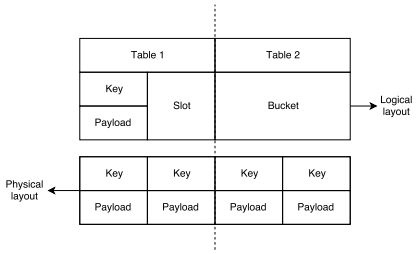

Fig. 2: Cuckoo hash-table structure

For other hashing techniques, we use a Structure of Arrays (SoA) for the hash table as shown in Figure 1. The hash tables have keys and payloads in different arrays with the same index pointing to a key and its corresponding index. We use this structure, as the payload is accessed only if the key is found in the hash table.

4.2 SIMD Accelerated Cuckoo Hashing

Ross et al., have given a detailed outline for performing SIMD accelerated probing in cuckoo hashing [14]. We extend their idea by adding the direct aggregation mechanism. We depict the general direct aggregation with probing in cuckoo hashing in Figure 3.

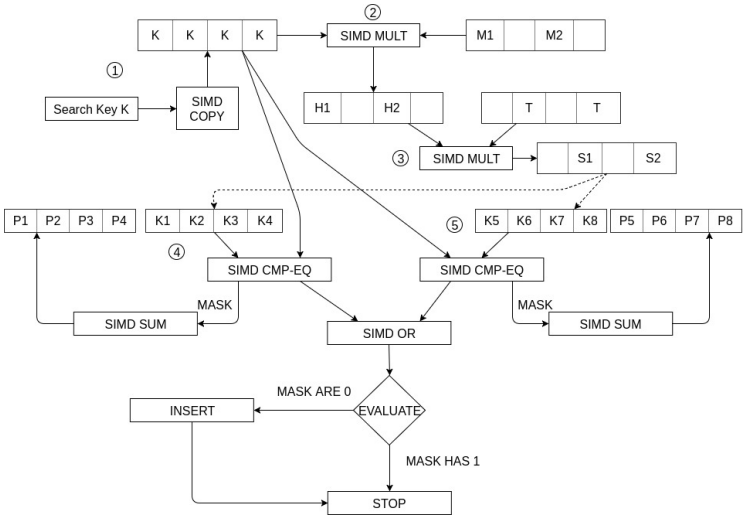

Fig. 3: SIMD accelerated cuckoo hashing (extended from Ross et al. [14])

Grouped aggregation using cuckoo hashing has two phases. First, the slot locations are identified using hashing functions. Second, we use the identified slots to probe the respective tables.

In our approach, the slots for a key are identified using multiplicative hashing. The function multiplies the key with a random multiplier, then performs modulo on the resultant with the table size to get the slots. To perform multiplicative hashing in SIMD, a SIMD COPY functions (e.g., _mm_set_epi32()) vectorize the search key. This key vector is multiplied with a multiplier vector using SIMD MULT functions (e.g., _mm_mul_epi32()) (2). Finally, the resultant vector is again multiplied with a table size vector (3). The slot values are available in the least significant bytes of the result vector.

Based on the slots, the bucket vector values are compared with the key vector using SIMD CMP-EQ functions (e.g., _mm_cmpeq_epi32()) providing a mask vector (4, 5). The mask vector based on the comparison has either 0 or 1 and this result along with the corresponding payloads are updated using SIMD SUM operations (e.g., _mm_add_epi32()). Finally, if all the masks are 0 then the key is inserted .

4.3 SIMD-Accelerated Linear Probing

Scanning hash table one at a time is time consuming. This is improved by using SIMD for scanning multiple slots in an instant.

In Figure 4, we describe the SIMD adapted linear probing mechanism. First, scalar multiplicative hashing function computes the slot for the given key. We use scalar function as only one slot is required. Second, the search key is vectorized using a SIMD COPY function (1). This search key vector is then compared with the values present in the pointed bucket using SIMD COMPARE

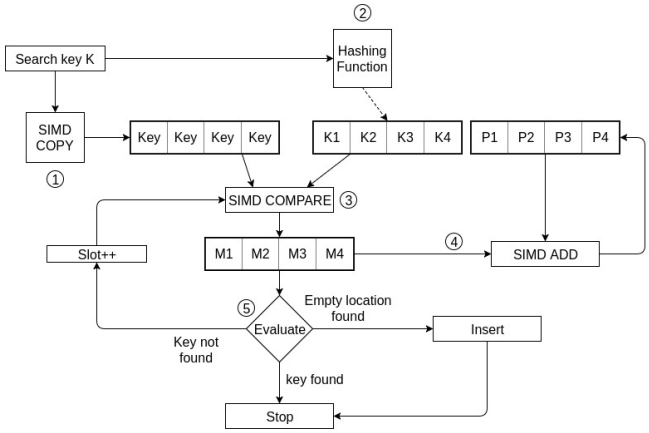

Fig. 4: SIMD accelerated linear probing

((e.g., _mm_cmpeq_epi32()). Final steps are similar to that of cuckoo hashing, where the resultant mask vector is added to the corresponding payloads (4) and insertion based on the mask is determined (5).

4.4 SIMD Accelerated Two-Choice Hashing

The major advantage of using SIMD for two-choice hashing is to compute the slots from two hash functions at a time. SIMD acceleration is similar to that of linear probing. Instead of comparing the key in a single vector of slots, we compare the vectors from the selected slots in sequence.

4.5 SIMD Accelerated Hopscotch Hashing

Hopscotch hashing is a multi-step algorithm, where the probing for key is done in the first and swapping of keys to have empty location in the second. Hence, this technique requires additional measures for adapting SIMD. In fact, it might even create an overhead for preprocessing the input to adapt SIMD for each of these steps. We call these steps forward and reverse probe. In the forward probe, we search for a key until the neighborhood boundary and empty space afterward. In reverse probe, we perform the swaps with the empty slots until we reach the neighborhood. We use SIMD to accelerate the probing of keys in the forward probe and swapping of keys while insertion.

We use the same SIMD acceleration of linear probing for the forward probe in hopscotch hashing (cf. Figure 4). For swapping of keys, we use the gather instructions in SIMD as given in Figure 5.

Forward Probe Within the neighborhood, direct aggregation is done similar to linear probing. Once outside the boundary, the table is probed for an empty location. This empty location index is used in the next phase for swapping of keys.

Reverse Probe A reverse probe is done to select the positions to swap the keys in order to have an empty location within the neighborhood boundary. During the reverse probe, the pointer moves back from the empty slot until a key inside the neighborhood of insert key can be swapped. On each step, the key to be swapped is stored into a swap array until the neighborhood of the insert key is reached. The keys are swapped for the indexes and the given key is inserted. If the neighborhood is not reached, then the table has to be rehashed.

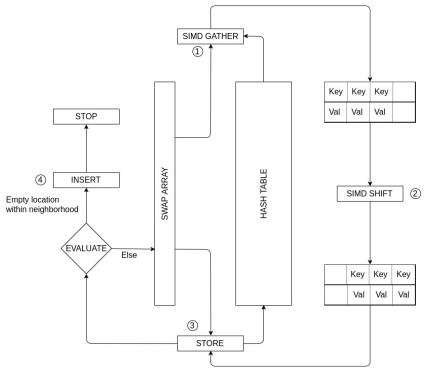

Fig. 5: SIMD accelerated hopscotch hashing - reverse probe

In the next section, we evaluate these SIMD-accelerated hashing techniques using different data distributions. We detail our evaluation setup first, followed by our results. Finally, we provide our insights on the results found.

5 Evaluation

To assess the efficiency of the presented hashing techniques, we measure their performance for different data distributions. In our hashing techniques, the insertion and aggregation functions are performed in a single atomic step. Hence, we include the overall insertion and probing time for determining the efficiency of the techniques. We conducted our experiments on a machine running CentOS Linux version-7.1.1503 and gcc version 4.8.5 with an octa core Intel Xeon E5-2630 v3s- 2014. The system is incorporated with the AVX2 instruction set.

5.1 General Assumptions and Experimental Setup

For all our experiments we have the below assumptions for terms of consistency. These are common across all the hashing techniques.

Key-payload pair: We assume that the key and payload to insert are 32-bit integers. Also, w.l.o.g., the value zero is not a valid key or payload value as it is used to represent empty slots in the hash table. In a production system, the maximum value in the domain can also be chosen.

Hash function: A multiplicative hashing function is used to disperse the input keys into different buckets. The main advantages are: (1) it can be parallelized

easily, (2) provides a good balance for arithmetic progression values (e.g., primary key). We use Knuth's multipliers in the hashing function [8].
Aggregation: For simplicity, we perform count as the aggregation function to be performed over the given set of keys. This can be easily extended to perform other aggregation operations such as max, min, sum. The approach fails for other dual pass functions such as average or standard deviation. In this case, normal two pass algorithm is used as in first pass total count is recorded followed by the respective function in the second pass.

All the experiments are run with an increasing number of keys and we record the CPU processing time for complete computation of grouped-aggregation for the different distributions. Every experiment is executed for 20 iterations and the results are averaged. We have performed two different tests over the hashing techniques. In Section 5.2, we discuss in detail the impact of the different parameters and we discuss in detail the impact of different distributions on these hashing functions.

The techniques are subjected to insertion with different distribution generators for our test cases. Each distribution represents the characteristics of the input keys provided to the hashing techniques. We synthesize these input based on descriptions given by Gray et al. [5]. The distributions used are (1) unique random, (2) uniform random, (3) moving cluster, (4) exponential, (5) self similar, and (6) heavy hitter. We use the parameters for the generation of the different distributions based on the values given by Gray et al. [5]. For unique random distribution, the seed for generation is selected based on the given input size. We use the random generator function to generate the keys for the uniform random generation. In exponential distribution, the lambda is set to as 0.5, where the number of keys is normalized. Our heavy hitter distribution produces 50% of the given input size as duplicates with remaining keys unique. Finally, the self similar distribution generates 80% of the given input size as duplicates and the remaining 20% is generated as unique keys.

5.2 Factors Affecting Cuckoo Hashing and Hopscotch Hashing

The performance of cuckoo and hopscotch hashing relies on the parameter values, swap number, and neighborhood size respectively. The maximum efficiency of these techniques is achieved by selecting the right value for these parameters. we vary these parameters and find their impact on the load factor. Load factor is the ratio of number of slots filled to the total number of slots available in a hash table. Since, both the hash tables in the worst case cannot accommodate 100% of the hash table slots, we use the parameter for which the load factor is the maximum.

Cuckoo Hashing We depict in Figure 6, how the swapping threshold impacts the achievable load factor. Since reaching the swap number threshold requires the table to be rehashed, the execution time of the hashing technique is directly related to the swap number and its corresponding load factor.

For this experiment, we iterate the insertion of keys with size equal to the size of the hash table. We vary the swap number in steps until the total key size. We are able to achieve a maximum load factor of 98.385% for swap number at 25% of the total number of keys. We were not able to achieve hundred percent of the load factor due to the dispersion of keys by the underlying hash function.

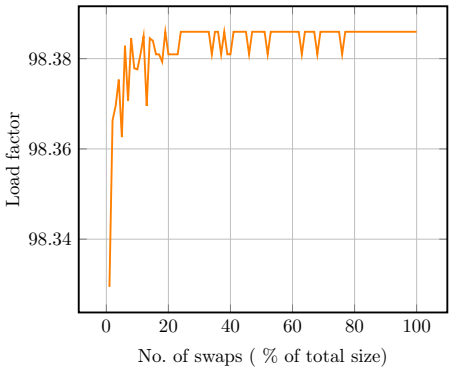

Fig. 6: Cuckoo hashing - swap threshold vs. load factor

Hopscotch Hashing Hopscotch hashing has minimal insertion time for a key within the neighborhood. Whereas, swapping is performed outside the neighborhood swapping for insertion. Thus, decreasing the neighborhood size leads to faster probing and slower insertion and increasing the neighborhood size does the vice versa. Hence, an optimal neighborhood size must be used for efficient execution. Similar to cuckoo hashing, we evaluate the optimal neighborhood of the hashing technique with different neighborhood sizes and the charts are plotted in Figure 7.

We run two tests to determine the correct neighborhood size for hopscotch hashing. In our first test, we determine the neighborhood size for optimal insertion time. We vary the neighborhood size from 10% to 100% the total size of the table and record the overall average insertion time for the different sizes. From the results plotted in Figure 7(a), we observe the lowest insert time is for a neighborhood size 20% of the total table size. We also see that the insert time increases rapidly after the neighborhood size 40% of the total table size mainly due to the multiple swaps for every insertion. Hence, a neighborhood size between 10% to 20% of the total size shows good performance for the hashing technique.

In our second experiment, we investigate the impact of neighborhood size on the load factor. In this experiment, we vary the neighborhood size and record the maximum load factor reached. This is plotted in Figure 7(b). From our observations, we are able to reach an average of 99% load factor. However, increasing the neighborhood size impacts in the runtime as the number of probe locations increases. We observe that a load factor of 98% is reached for neighborhood size after 20% of the total size for all the data sizes. Hence, the best neighborhood size lies between 20% to 30% of the total size. The remaining keys are not inserted due to poor dispersion of the keys by the hashing function.

Summary Using the above mentioned parameters, we could reach a maximum load factor of 98% for these two hashing techniques after which re-hashing of the tables is probable. We set the number of swaps for cuckoo hashing as 25 and the neighborhood size of hopscotch hashing as 50 for all the experiments below. Also, we keep the load factor as 96%.

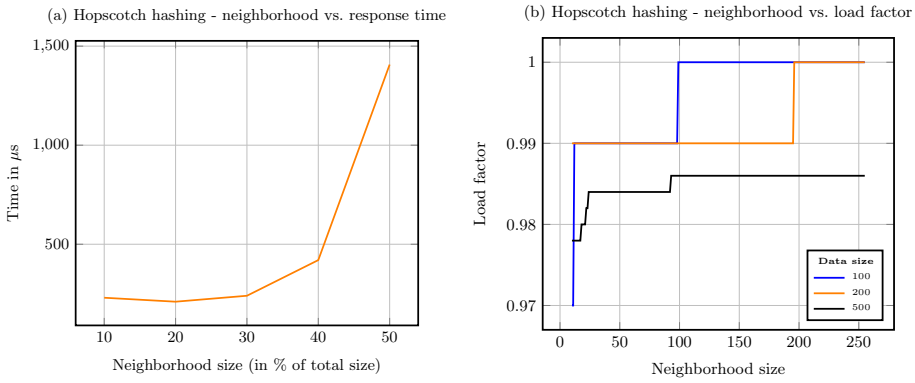

Fig. 7: Hopscotch hashing - impact of neighborhood

Impact of Different Distributions In this experiment, we aggregate keys of various distributions using the hashing techniques with their optimal parameters. Since cuckoo hashing is having a maximum load factor of 98%, we set our maximum load factor as 95% in order to insert all the given keys. Also, we use the optimal parameters for cuckoo and hopscotch hashing.

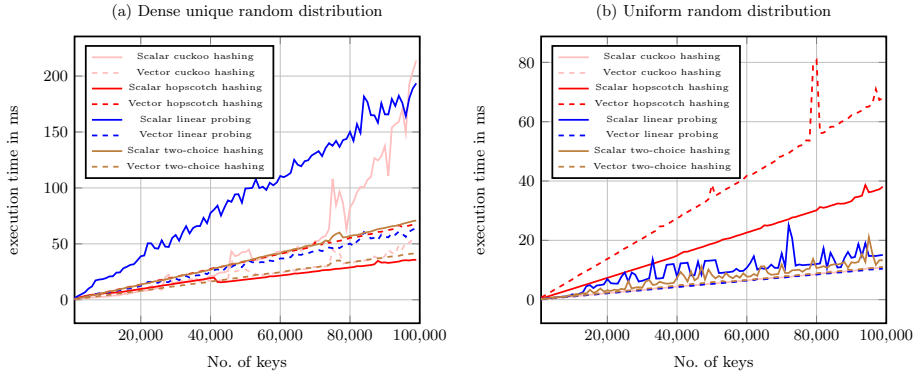

Fig. 8: Hopscotch hashing - Impact of distribution

Dense random unique distribution: For unique random distribution linear probing performs worst, whereas the vectorized version competes with other techniques as shown in Figure 8(a). This is mainly due to the advantage of probing multiple slots in a single step. Cuckoo hashing performance degrades with increasing data size, mainly due to swaps during insertion. The other hashing techniques have a linear increase of runtime as they have no additional overhead in inserting keys. Specifically, hopscotch hashing works efficiently for this distribution due to the neighborhood size reducing the number of keys to be probed.

Uniform distribution: Figure 8(b) shows the performance graph for uniform distribution. Hopscotch hashing performs worse in this case due to the overhead of swapping. Cuckoo hashing has nearly similar efficiency as the hopscotch hashing, due to again the penalty from insertion. In case of linear probing and two-choice hashing, the efficiency depends on the order of hashing keys, as the best order of insertion needs less probing. Hence, the runtime oscillates from low to high. However, the vectorized version of the hashing techniques has near linear runtime, except for hopscotch hashing. All other vectorized versions are more efficient than their scalar version, with linear probing having the best speed-up of 3x the scalar version mainly due to an early detection of an empty slot during the probe.

Moving cluster: We see a peculiar impact on cuckoo hashing for the moving cluster distribution shown in Figure 9. It performs efficiently until it reaches 50K and after that its performance degrades rapidly. This is mainly due to the heavy swapping of keys inside the cluster. Whereas, the other hashing techniques have a linear performance with increasing data size.

Other distributions: We omit a performance graph for the other distributions, as the technique's behavior are similar to what we see for uniform distributed values. The only notable difference is that vectorized cuckoo has good performance for self-similar and exponential distributions for earlier data sizes but is soon surpassed by vectorized two-choice hashing.

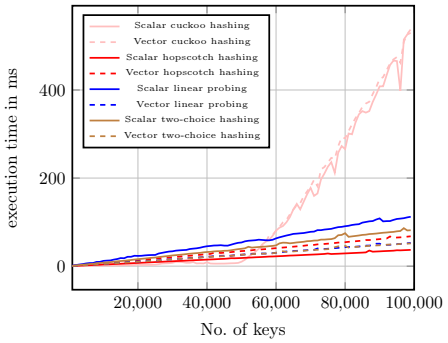

Fig. 9: Moving cluster distribution

Speed-up Gain Based on our results, we found that several factors influence the speed-up gained by vectorization of the hashing techniques. We plot the speed-up of the hashing techniques in Figure 10.

We observe from the figure, linear probing (LP) has consistently positive impact of SIMD acceleration with the maximum of up to 3.7x the speed of scalar version. Two-choice hashing (TCH) also has a considerable impact of vectorization with the maximum gain of 2.5x, but the speed-up depends on the input data distributions. SIMD vectorization has no impact on cuckoo

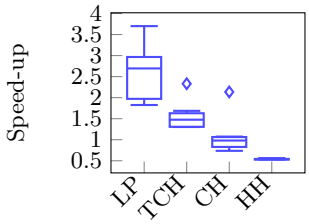

Fig. 10: Speed-up across all tested distributions

hashing (CH) as the insert time for cuckoo hashing balances the speed-up gained due to probing of keys. However, for distributions with a high number of duplicate keys, the technique gains a speed-up of 2x the scalar version. Finally, our result indicates that the usage of SIMD instructions for hopscotch hashing (HH) is not providing a performance increase. This is due to the preprocessing steps needed for inserting keys. Mainly for SIMD insertion, single key insertion re-arranges multiple keys inside the table.

6 Conclusion

Vectorization impacts the execution of a grouped aggregation function. This impact differs based on the hashing technique used to compute the results. In the work, we explored the impacts on vectorizing the commonly used open-addressing hashing techniques.

For vectorization of the techniques, we provide a framework with horizontal vectorization along with an interleaved insertion. In our method, a given key is searched in a hash table using vectorized probing with either an insertion of a key or an update of the aggregate payload. We detail the execution flow of vectorized hashing techniques and discuss the complexities in incorporating them.

In our experiments, we found that the overhead of vectorizing a scalar key limits the overall performance gain from SIMD. For the case of linear probing and two choice hashing, SIMD acceleration provides gain of 2x with respect to their scalar implementation. Whereas in case of cuckoo hashing and hopscotch hashing, we get negative impact from SIMD. This is mainly due to additional overheads for key insertions in these techniques.

Using our framework, we provide a possible vectorization model for hashing techniques. This model is extensible for further techniques as well as other vectorization strategies. Our current framework is not scalable for a multi-CPU system but a synchronization mechanism can be easily added. Finally, from our analysis, we found that the hashing technique related parameters must be tuned for efficient execution.

7 Acknowledgments

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

References

1. Broneske, D., Meister, A., Saake, G.: Hardware-sensitive scan operator variants for compiled selection pipelines. In: Datenbanksysteme für Business, Technologie und Web (BTW). pp. 403–412 (2017)
2. Broneske, D., Saake, G.: Exploiting capabilities of modern processors in data intensive applications. it - Information Technology **59**(3), 133 (2017)
3. Cieslewicz, J., Ross, K.a.: Adaptive aggregation on chip multiprocessors. Proceedings of the Very Large Databases (VLDB) pp. 339–350 (2007)
4. Flajolet, P., Poblete, P., Viola, A.: On the analysis of linear probing hashing. Algorithmica **22**(4), 490–515 (1998)
5. Gray, J., Sundaresan, P., Englert, S., Baclawski, K., Weinberger, P.J.: Quickly generating billion-record synthetic databases. International Conference on Management of Data (SIGMOD) pp. 243–252 (1994)
6. Herlihy, M., Shavit, N., Tzafrir, M.: Hopscotch hashing. In: International Symposium on Distributed Computing. pp. 350–364. Springer (2008)
7. Jiang, P., Agrawal, G.: Efficient SIMD and MIMD Parallelization of Hash-based Aggregation by Conflict Mitigation. Proceedings of the International Conference on Supercomputing (ICS) pp. 24:1–24:11 (2017)
8. McClellan, M.T., Minker, J., Knuth, D.E.: The Art of Computer Programming, Vol. 3: Sorting and Searching. Mathematics of Computation **28**(128), 1175 (1974)
9. Pagh, R., Rodler, F.F.: Cuckoo hashing. Journal of Algorithms **51**(2), 122–144 (2004)
10. Polychroniou, O., Raghavan, A., Ross, K.A.: Rethinking SIMD Vectorization for In-Memory Databases. Proceedings of the International Conference on Management of Data (SIGMOD) pp. 1493–1508 (2015)
11. Polychroniou, O., Ross, K.A.: High throughput heavy hitter aggregation for modern SIMD processors. Proceedings of the Ninth International Workshop on Data Management on New Hardware (DaMoN) pp. 6:1–6:6 (2013)
12. Richa, A.W., Mitzenmacher, M., Sitaraman, R.: The power of two random choices: A survey of techniques and results. Combinatorial Optimization **9**, 255–304 (2001)
13. Richter, S., Alvarez, V., Dittrich, J.: A seven-dimensional analysis of hashing methods and its implications on query processing. Proceedings of the Very Large Databases (VLDB) **9**(3), 96–107 (2015)
14. Ross, K.A.: Efficient hash probes on modern processors. In: Proceedings of the International Conference on Data Engineering (ICDE). pp. 1297–1301. IEEE (2007)
15. Zhou, J., Ross, K.A.: Implementing Database Operations Using SIMD Instructions. In: International Conference on Management of Data (SIGMOD). p. 145 (2002)