Efficient SIMD and MIMD Parallelization of Hash-based Aggregation by Conflict Mitigation

Peng Jiang, Gagan Agrawal
Department of Computer Science and Engineering
The Ohio State University
Columbus, OH 43210, USA
{jiang.952,agrawal.28}@osu.edu

ABSTRACT

As the rate of data generation is growing exponentially each year, data aggregation has become one of the most common and expensive operations for data analysis. Previous efforts to accelerate data aggregation have been mainly focused on multi-core CPUs, improving single-core cache performance and/or reducing multi-core data synchronization overheads. In this paper, we aim at utilizing both SIMD and MIMD of a modern processor with a more recent (and wide lane) SIMD instruction set. We find that a straightforward method for vectorization of hash table often cannot deliver good performance, because wider SIMD vector increases data conflicts among the lanes (especially with skewed data). To address this problem, we design a variant of basic bucket hashing and a *bucketized* aggregation procedure that can utilize both SIMD and MIMD parallelism efficiently. Our approach first adds distinct offsets to input rows on different SIMD lanes, which reduces the possibility of different lanes accessing identical slot in the hash table. An efficient bucketized aggregation procedure is invoked to save space when the hash table is saturated, or to calculate the final results after all input rows have been inserted into the hash table. For parallelization across cores, we adopt separate hash tables and optimize with parallel reduction and a hybrid approach. We evaluate our methods with input datasets of different distributions. On a single core of Intel Xeon Phi, we obtain 1.6x to 2.9x speedup (over serial code) using our SIMD approach, and outperform a straightforward SIMD implementation by up to 7x. Over multiple cores, our approach has a near-linear scalability.

ACM Reference format:
Peng Jiang, Gagan Agrawal. 2017. Efficient SIMD and MIMD Parallelization of Hash-based Aggregation by Conflict Mitigation. In *Proceedings of ICS '17, Chicago, IL, USA, June 14-16, 2017,* 11 pages.
DOI: http://dx.doi.org/10.1145/3079079.3079080

1 INTRODUCTION

In the 'big-data' era, there is a critical need to quickly aggregate and summarize vast amounts of information. Data aggregation has become the cornerstone of many important technologies such as SQL, MapReduce, OLAP cubes, and statistical languages [4, 8]. An example of data aggregation is a `GROUP BY` clause in SQL where the rows are grouped on a specified condition and an aggregation function(s) is performed on each group. Although it is seemingly easy to implement, aggregation is one of the most expensive data analytics operators, and can be a bottleneck when applied to large input streams.

Many efforts have been made over the last two decades to accelerate aggregation on traditional multi-core CPUs [5, 6, 22]. There are mainly two approaches: *hash-based* and *sort-based*. *Hash-based aggregation* is a widely researched and implemented algorithm, which inserts input rows into a hash table, using the grouping attributes as the key, and aggregates the remaining attributes in-place. Another algorithm for aggregation is the *sort-based* method. This method sorts the input, so that the rows with identical keys are placed consecutively, and then aggregates the (consecutive) rows with identical keys to obtain the results. Both of these two algorithms have their advantages and disadvantages. Hash-based aggregation has a lower computation complexity ($O(n)$). Moreover, though a recent study has shown that sort-based aggregation are equivalent to hash-based aggregation in terms of cache efficiency [18], hash-based aggregation is believed to have better performance when the number of groups is small (such that the output fits in the cache) [1, 15, 18].

While parallelization of data aggregation has been widely studied over multi-cores, to-date the research on exploiting another form of parallelism – the SIMD parallelism – is quite limited. Zhou *et al.* [24] used the early Intel SIMD instructions (SSE) to accelerate data analytics operations. Polychroniou *et al.* [20] proposed a vectorization technique for aggregating *heavy-hitter* data using the early SIMD instructions. These early efforts were constrained by the short SIMD vector (containing only four lanes) and inflexible SIMD instructions (i.e., requiring contiguous memory loads with further restrictions on alignment).

1.1 Existing Work and Limitations

In recent years, SIMD instruction sets have become more flexible. Particularly, Intel Xeon Phi features wider SIMD vectors and more flexible SIMD instructions. There has been an increasing level of interest in extracting this fine-grain parallelism for data-intensive applications. Polychroniou *et al.* [19] take advantage of these new SIMD features and propose improved vectorization method for in-memory data analytics operations. Jha *et al.* [12] utilize both SIMD and MIMD parallelism of an Intel Xeon Phi to accelerate main memory hash joins.

In reviewing the existing SIMD vectorization methods for hash-based aggregation, we can see why it is still challenging to achieve good performance. Broadly, in the existing methods, multiple keys are *gathered* into a SIMD vector where their hash addresses are calculated simultaneously. Before any aggregation or writing, the program needs to check if there are *data conflict(s)* among SIMD lanes. These conflicts arises because multiple writes to a single memory location can lead to incorrect results. If there is a conflict, only one of the conflicting lanes can update the values in the hash table, and the rest of lanes need to wait till the next round of processing.

ICS '17, Chicago, IL, USA

DOI: http://dx.doi.org/10.1145/3079079.3079080

More specifically, for hash-based aggregation, a data conflict occurs when multiple lanes have an identical hash value (or end up at a same location after several steps of probing, as we will discuss later). The problem gets more serious when data is skewed. Examples of such situations include a *heavy-hitter* input, where one value accounts for a large percentage of the group-by keys, or in a *Zipf distribution* input where a small amount of values occur much more frequently than others in the group-by keys. In such cases, the chances of identical keys within 16 consecutive rows are high.

A common technique to resolve data conflicts in irregular (data-dependent) applications is to use preprocessing or an *inspector*, which has been applied for SIMD processing also [3, 13]. However, these approaches work only when the cost of the inspector can be amortized. Such amortization is possible only when the application involves many iterations where the data access pattern is the same or very similar. In hash-based aggregation, this is not the case because one takes only one pass through the input. In summary, while data aggregation has abundant data-level parallelism, nature of real-world data, coupled together with a lack of appropriate hardware support to handle conflicts, make efficient SIMD parallization extremely hard [8].

1.2 Our Contributions

This paper focuses on exploiting SIMD (and MIMD) parallelism for hash-based aggregation, with an emphasis on avoiding conflicts without any reorganization of data. We propose a variant of *basic bucket hash table*, together with a *bucketized aggregation procedure*. As in bucket hashing, we divide the hash table into buckets of 16 slots, and the hash value of an input row is the index of the bucket it goes to. Unlike basic bucket hashing, however, we add different offsets ([0, 1, 2, ..., 15]) to keys on different SIMD lanes as their initial addresses. It ensures that none of the lanes have identical initial address while constraining the input row in the bucket. Based on this data structure, we devise a bucketized aggregation procedure that performs aggregation within each bucket. This procedure is invoked when a key cannot find a match or an empty slot in a bucket, or at the end of the execution to calculate the final results. For parallelization over cores, we adopt the separate hash table method [5] and optimize it with two techniques: a parallel reduction technique that minimizes the overhead of merging the separate tables, and a hybrid hash table technique that reduces memory consumption when the hash table is out of cache and many threads are used.

We evaluate our methods using a set of datasets chosen to have different distributions. On a single core of Intel Xeon Phi, our approach achieves 1.6x to 2.9x speedup over the serial code, and outperforms a straightforward SIMD implementation by up to 7x. On multiple cores, our approach has a near linear scalability in most of the cases.

2 BACKGROUND

This section first describes the SIMD instruction set we target. Next, we provide background on hash-based aggregation and its straightforward vectorization.

2.1 SIMD Instruction Set

The SIMD instruction set we target is one currently supported in Intel Xeon Phi. The SIMD vectors are of 512 bits, which means 16 integers (or 16 single precision floating point) can be processed simultaneously in a vector. The wide SIMD vector is further empowered by sophisticated SIMD instructions such as the *gather/scatter* and the *mask* operations. The *gather/scatter* operations can load and store data at non-contiguous memory addresses that are specified by an *index vector*. The *mask* operations, including *mask-read*, *mask-write*, *mask-add*, and *mask-compare*, allow the program to apply an operation on a specified subset of lanes within a SIMD vector. Table 1 lists the SIMD instructions that are important for our discussion. The instructions listed here are abbreviations for the *Intel Intrinsic*, which is the interface provided by Intel ICC compiler to use the SIMD features on Intel (co)processors.

Algorithm 1 Serial_Aggregation ($keys$, $payloads$)

```
1:  for (i = 0; i < num_rows; i++) do
2:      key = keys[i]
3:      payload = payloads[i]
4:      address = Hash(key)
5:      match = false;
6:      while (hash_table[address]! = empty) do
7:          if (key == hash_table[address]) then
8:              Aggregate(payload, hash_table, address)
9:              match = true;
10:             break
11:         end if
12:         address++
13:     end while
14:     if (!match) then
15:         hash_table[address] = key
16:         WriteTo(payload, hash_table, address)
17:     end if
18: end for
```

2.2 Hash-based Aggregation and Its Naive Vectorization

The basic idea of hash-based aggregation is to insert the inputs into a hash table by the *key(s)* and aggregate one or more attributes (called the payload). Algorithm 1 shows the serial code of hash-based aggregation using a *linear probing* hash table. Linear probing involves traversing the hash table linearly to find either a match or an empty slot. The serial code first reads the key and the payload of an input row and calculates its initial address in the hash table by giving the key to a hash function. Then, it probes the hash table starting from the initial address for a match or an empty slot. If a match is found before reaching an empty slot, the payload is aggregated with the values in that match slot. Alternatively, if no match is found, the payload and the key is directly written to the empty slot.

Now, we consider vectorization of this method. To the best of our knowledge, there is no work specifically on hash-based aggregation with the new SIMD instruction set. However, the work on vectorizing the steps of building and probing a hash table [12, 19] can be adopted here. We refer to it as `Naive` vectorization of hash-based aggregation and will use it as a baseline for comparison with our approach. For simplicity, we only discuss the *linear probing* hash table here. The reason is that hash table occupancy is not a critical issue for aggregation. Sophisticated hash tables with multiple hash functions such as double hashing and cuckoo hashing may have better occupancy, but also have more overheads for computing hash values, and thus are not helpful for accelerating aggregation.

Algorithm 2 shows the naive vectorization of hash-based aggregation. Here we assume that both keys and payloads are 32 bit integers (algorithms for other data types are similar). An outer loop iterates over all the input rows, and in each iteration, the input rows are loaded and processed. The $free$ mask indicates the lanes that have

SIMD Instruction	Description
`_mm_gather(idx, base)`	Load data from positions specified by `idx` starting from `base`
`_mm_scatter(idx, base, val)`	Store `val` to positions specified by `idx` starting from `base`
`_mm_mask_gather(def, mask, idx, base)`	Load data on the lanes with set `mask`, others are assigned to `def`
`_mm_mask_scatter(mask, idx, base, val)`	Store `val` on the lanes with set `mask`
`_mm_cmp**(va, vb)`	Compare two vectors and return the result as a mask
`_mm_add(va, vb)`	Add two vectors
`_mm_mul(va, vb)`	Multiply two vectors
`_mm_mask_add(def, mask, va, vb)`	Add two vectors on the lanes with set `mask`, others are assigned to `def`
`_mm_mask_mul(def, mask, va, vb)`	Multiply two vectors on the lanes with set `mask`, others are assigned to `def`

Table 1: SIMD instructions used in this work

Algorithm 2 Naive_SIMD_Aggregation (*keys*, *payloads*)

```
1:  __mmask16 free, notempty, match, nonconf, probe
2:  __m512i vkey, vpayload, vload, vhkey, vaddress
3:  initialize(hash_table)
4:  free = 0xFFFF
5:  for (i = 0; i < num_rows; i+=Nbits(free)) do
        ▷ Step 1: load new input rows into free SIMD lanes
6:      vload = load_indices[free]
7:      vkey = _mm_mask_gather(vkey, free, vload,
        keys + i)
8:      vpayload = _mm_mask_gather(vpayload, free, vload,
        payloads + i)
        ▷ Step 2: probe for match or empty slot
9:      vaddress = Hash(vkey)
10:     vhkey = _mm_gather(vaddress, hash_table)
11:     notempty = _mm_cmpneq(vhkey, -1)
12:     match = _mm_cmpeq(vhkey, vkey)
        ▷ Step 3: wait until all lanes find match or empty slot
13:     while (notempty! = match) do
14:         probe = ~match&notempty
15:         vaddress = _mm_mask_add(vaddress, probe,
            vaddress, 1)
16:         vhkey = _mm_gather(vaddress, hash_table)
17:         notempty = _mm_cmpneq(vhkey, -1)
18:         match = _mm_cmpeq(vhkey, vkey)
19:     end while
        ▷ Step 4: check conflicts among SIMD lanes
20:     nonconf = CheckConflicts(vaddress)
        ▷ Step 5. update non-conflict lanes
21:     _mm_mask_scatter(hash_table, ~notempty
        &nonconf, vaddress, vkey)
22:     Aggregate(match&nonconf, vpayload)
23: end for
```

been processed in the previous iteration and can be filled with new input rows. Initially, $free$ equals $0xFFFF$, which means all the lanes can be loaded with new input rows. The *Nbit* procedure, which returns the number of bit-1 in a mask, is used to calculate the read position in the input. Processing of each vector of elements follows five steps, which are marked through the comments in Algorithm 2.

Step 1: Multiple keys are loaded into an input key vector, and their hash addresses are computed simultaneously. Since the input rows on some of the lanes may not be processed in the previous iteration due to conflicts, new input rows are only loaded into lanes whose records were processed.

Step 2: The values at the hash addresses are *gathered* from the hash table to another SIMD vector, with the goal of finding matches or empty slots. If the value on a lane matches the value on the corresponding lane of the input key vector, then the new payload on this lane is ready to be aggregated with the value in the hash table. If the value on a lane is empty, the new input row can be stored in the empty slot. For the lanes where we do not find a match or an empty slot, we need to probe linearly by adding 1 to their addresses.

The SIMD lanes containing matches or empty slots can either start processing immediately or wait until all of the lanes find a match or an empty slot. We have experimented with both approaches and determined that using an inner loop to find matches or empty slots for all SIMD lanes before processing is more efficient, so we adopt this approach as *Step 3* in our baseline. Note that this is in contrast to Polychroniou *et al.*'s conclusion for simple hash table building and probing that SIMD lanes with matches or empty slots should start processing immediately and be filled with new input rows as soon as possible [19]. Aggregation involves more computation in each iteration of the outer loop. Thus, using an inner loop to find a match or an empty slot for all SIMD lanes is more efficient.

Step 3: An inner loop keeps incrementing the addresses on the lanes that do not find a match or an empty slot until all of the lanes find a match or an empty slot.

Step 4: The program checks conflicts among the SIMD lanes. Only one among the set of conflicting lanes is considered valid and can update the value to the hash table. The rest of lanes within the set need to be invalidated until next round of processing. Polychroniou *et al.* [19] provide a technique to detect conflicting lanes in the address vector, which we use here. First, a vector of unique values per lane (e.g., $[0, 1, 2, \ldots, 15]$) is scattered to the positions specified by the address vector. Then, the values are gathered again from the same addresses. If the scattered value does not match the gathered value, it means the value on that lane has been overwritten by another lane. The lanes on which the scattered value matches the gathered value are safe to update – this is because they either have no conflict or are the last lane among the set of conflicting lanes. Note that this functionality may be directly supported in the hardware in the future (*vpconflictd* in AVX3), but currently available instruction sets will require such a method to be explicitly implemented.

Step 5: The value on the non-conflicting lanes are updated. Specifically, the lanes finding a match aggregate the new payloads to the value in the hash table, while the lanes finding an empty slot stores the new keys/payloads in the empty slots.

3 MITIGATING CONFLICTS IN SIMD PROCESSING

In this section, we first show how a naive approach to SIMD parallelization of hash-based aggregation is negatively impacted by data

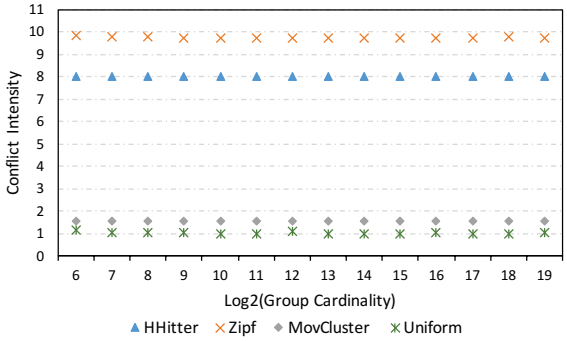


Figure 1: Conflict intensities of data of different distributions and different group cardinalities

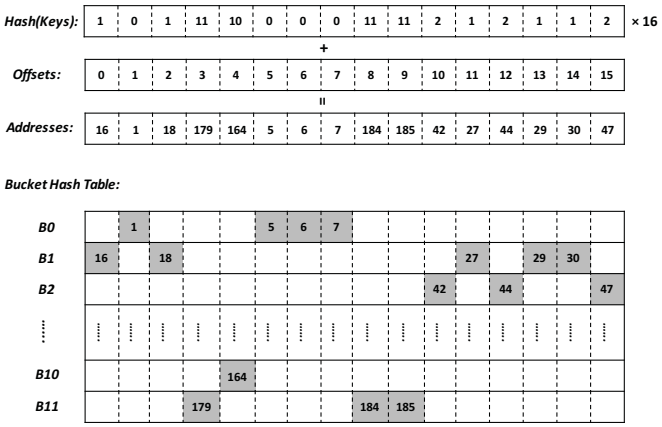


Figure 2: An example of bucket hashing with offsets

conflicts among the SIMD lanes. We then present a technique that can, to a large extent, mitigate these data conflicts at runtime, and without requiring expensive preprocessing or reorganization. Our technique is based on two major ideas: *bucket hashing with offsets* and a *bucketized aggregation procedure*. We allow duplicate keys in the hash table, but the duplicate keys are confined in a bucket such that the final results of a key can be easily calculated by aggregation within that bucket.

3.1 Conflicts in Naive Method

Conflicts among SIMD lanes can significantly impair performance of Algorithm 2 as each conflicting lane adds one more iteration to the outer loop. We now define a measure of conflict intensity in data. Suppose there are n rows in a dataset, and the rows are divided into $n/16$ blocks (denoted as b) of 16 consecutive rows. Let r be the row that has the highest number of conflicts within the block i. Now, if there are c_i rows conflicting with r in block i, the number of iterations to process this block with SIMD is at least $1 + c_i$. We define the intensity of conflicts in an input, $IConf$, to be the ratio of the number of SIMD processing iterations to the number of blocks, i.e.,

$$Iconf = \frac{\sum_{i=0}^{b-1}(1 + c_i)}{b}$$

$Iconf$ also turns out to be the reciprocal of SIMD utilization. If there are no conflicts in any block, then $c_i = 0$, $Iconf$ is 1, and the SIMD utilization is 1 as all of the 16 lanes in SIMD vector are processed simultaneously. In the worst case when all of the rows in the input have the same key, $c_i = 15$, $Iconf$ becomes 16, and the SIMD utilization is $1/16$ as Algorithm 2 processes one row per iteration. Strictly speaking, $IConf$ is a lower bound because even two different keys in a block can be hashed to an identical slot in the hash table, creating a conflict even when keys are not identical.

We generate data of four different distributions, which we denote as *heavy hitter*, *Zipf*, *moving cluster*, and *uniform*, and evaluate their conflict intensities. In the *heavy hitter* input (`HHitter`), one value account for 50% of the group-by keys, while the other values are chosen uniformly from the other group-by keys. The Zipf[1] uses an exponent of 0.5. In the moving cluster input (`MovCluster`), the size of the window of data locality (which gradually shifts) is 64. The value in the uniform input are chosen randomly and uniformly from all of the group-by keys.

[1] http://www.hpl.hp.com/research/idl/papers/ranking/ranking.html

Figure 1 shows the $IConf$ of data of the four distributions and different group cardinalities. The uniform data has conflict intensities of near 1, which means conflicts in this data are rare and have little effect on the performance of SIMD processing. The conflict intensities of moving cluster data are about 1.5, which means every two blocks of 16 input rows cost three iterations of the outer loop in Algorithm 2 on the average. A large amount of conflicts exist in heavy hitter input. The conflict intensities of heavy hitter data are about 8, which means it takes eight iterations of the outer loop in Algorithm 2 to process one block of 16 rows in the input. The Zipf input has the largest conflict intensities of near 10 among the four distributions. The results validate that the data conflicts can significantly slow down the naive SIMD processing of data aggregation.

3.2 Vectorized Bucket Hashing with Offsets

We now present a method that addresses the conflicts in the naive method.

As a background, consider the *basic bucket hashing*, where a hash table is divided into buckets, each of which has multiple slots. When inserting a record, the key is first hashed to determine the bucket it belongs to, and then, this bucket is probed to find an empty slot. Though an infrequent situation, if the bucket is full, the record will be stored in an overflow bucket – a large capacity store that is shared by all of the buckets in the hash table. The records in the overflow bucket will be aggregated by a simple sort-based method. Similarly, when searching for a record, the key is first hashed to determine the bucket the record resides in, and then a search is performed within that bucket. If the key is not found and the bucket still has a free slot, then the search is complete. However, if the bucket is full, the search continues to the overflow bucket.

Now, consider using the bucket hash table for data aggregation. An input row is first hashed to a bucket, and then, search is performed within the bucket to find a match or an empty slot. If a match is found, the payload is aggregated with the record in that slot; if an empty slot is found, the input row is written directly to that slot. If the bucket is full and none of the records in the bucket matches with the input row, the input row is added to the overflow bucket. The records in the overflow bucket need to be aggregated at the end of the processing.

The vectorization of bucket hash table based aggregation is similar to Algorithm 2. The difference is that the third step in the algorithm needs to search within a bucket. This is implemented by checking if

there is any address in the vector $vaddess$ that exceeds the end of its bucket. If so, the address is reset to the start of the bucket.

This algorithm is still negatively impacted by data conflicts among the SIMD lanes. However, based on the basic bucket hashing, this problem can be easily solved by adding different offsets to different SIMD lanes. Specifically, the number of slots in each bucket is configured to be the number of lanes in the SIMD vector, say W. The keys are given to the basic bucket hash function, and the output addresses are all at the start of each bucket. Next, the address on each lane is incremented by an unique offset less than W (i.e., to the address vector $vaddress$ in Algorithm 2 we add an offset vector $[0, 1, 2, \ldots, W-1]$).

Now, there are two important properties of this offset vector: 1) it keeps the addresses within their original hashed buckets, and 2) when two keys are hashed into an identical bucket, it distributes them to different slots within the bucket. The first property ensures the correctness of bucket hashing, while the second property guarantees that all of keys in the SIMD vector have different initial addresses for probing. Actually, the bucket number indicates the row a key goes to, while the lane number of that key in the SIMD vector is its column number in the bucket. Figure 2 shows an example of a vector of keys being hashed into a bucket hash table with offsets. The hash table has 12 buckets with each containing 16 slots. The hash function first generates the bucket number for each key, and then, the initial probing address is calculated by adding the offset vector to the product of the bucket number and the bucket size. All of the keys have an unique probing address in the hash table.

The new vectorization approach has two advantages over Algorithm 2 for data aggregation. First, it allows duplicate keys in the hash table, which increases parallelism of the computation – distinct lanes of a vector can aggregated into the same key simultaneously. In comparison, Algorithm 2 allows only one of the identical keys to be aggregated in one iteration of the outer loop. Also, because a cache line (of the same length as the SIMD vector) is the basic unit of data movement, having multiple keys also improves the spatial locality of the computation if a SIMD vector contains multiple identical keys. The downside of duplicate keys is that it consumes proportionally more space in the hash table. However, a simple *bucketized* aggregation procedure, which will be described later, can save space for new keys.

The second advantage of the new vectorization approach is that the initial probing addresses in a SIMD vector are all different. Although it is still possible that two lanes end up at the same location after probing (so the step for conflict checking in Algorithm 2 cannot be bypassed), it is much less likely to happen compared to a procedure that starts with identical addresses from each SIMD lane. This significantly decreases data conflicts among the SIMD lanes, and thus, increases the SIMD utilization.

3.3 Bucketized Aggregation

Consider the process of aggregation using the bucket hashing with offsets. Once a bucket is full and no match is found for an input key, the program needs to aggregate the records in that bucket and create space for the input key. This procedure is called a *bucketized aggregation*. Figure 3 shows an example of bucketized aggregation taking place. At some point during the hashed-based aggregation, an input key, say 9, is mapped to bucket i which has no empty slot and none of the keys in bucket i is equal to 9. As show in Figure 3a, there are multiple slots in the bucket that have identical keys. These identical keys can be merged into a single slot without losing the correctness of the computation as long as the values are merged

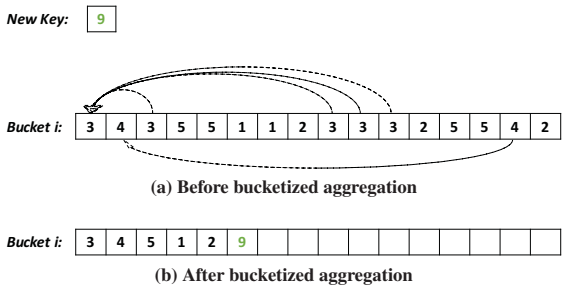


Figure 3: An example of bucketized aggregation

Algorithm 3 Bucket_Aggregation ($hash\_table$, $bucket\_id$)

```
1:  addr = StartingAddress(bucket_id)
2:  for (pivt = addr; pivt < addr + W − 1; pivt + +) do
        ▷ If pivot slot is empty, find a non-empty slot to fill it
3:      if (hash_table[pivt] == empty) then
4:          for (j = pivt + 1; j < addr + W; j + +) do
5:              if (hash_table[j]! = empty) then
6:                  hash_table[pivt] = hash_table[j]
7:                  hash_table[j] = empty
8:                  MoveValue(pivt, j)
9:                  break
10:             end if
11:         end for
            ▷ The procedure returns if no non-empty slot
12:         if (j == addr + W) then return end if
13:     end if
        ▷ Aggregate matching slots into pivot slot
14:     for (j = pivt + 1; j < addr + W; j + +) do
15:         if (hash_table[pivt] == hash_table[j]) then
16:             hash_table[j] = empty
17:             Aggregate(pivt, j)
18:         end if
19:     end for
20: end for
```

together with the keys. For example, as shown by the arrow in Figure 3a, all of the slots with a key 3 can be merged into the first slot, and the same process can be done for key 4, key 5 etc (we only show the arrows for key 3 and key 4 in the figure). Figure 3b shows the slots in bucket i after the procedure of bucketized aggregation. Every identical key occupies only one slot in the bucket, creating some empty slots for new input keys. Now, the input key 9 can be inserted to an empty slot in bucket i.

Algorithm 3 gives an implementation of this procedure. Here, W is the number of slots in each bucket. Starting from the first slot in the bucket, the procedure matches the slots following the *pivot slot*. Once a match is found, the value in the matching slot is merged with the pivot slot, and the matching slot is now empty. After all of the following slots have been examined, the pivot moves to the next slot in the bucket.

During the execution, the pivot slot can be empty due to a previous aggregation. If so, the procedure first searches for a following non-empty slot – if such a slot is found, this non-empty slot is set empty with its key moved to the pivot slot. If there is no non-empty slot following the pivot slot, the procedure completes as all of the records

are ahead of the pivot slot. Note that such *in-bucket* aggregation does not affect the correctness of the program because our hashing method ensures that all identical keys are stored within a bucket.

If implemented sequentially as shown in Algorithm 3, the bucketized aggregation will take $W * W$ steps to complete all the merging. As W is limited and this procedure does not take place very often, it does not incur much overhead to the computation. Moreover, Algorithm 3 can be accelerated by SIMD. The inner loop of Algorithm 3 is actually comparing all of the keys in following slots with the pivot key, which can be done by a single SIMD comparison instruction. The slots with an identical key to the pivot key are merged by a single horizontal-reduction instruction which is supported in Xeon Phi. This reduces the steps of merging a bucket to W. With future vector instructions such as *vpconflictd* in AVX3, the outer loop of this procedure, which finds the identical keys in the bucket, can also be done by a single SIMD instruction, further reducing the overhead of this procedure.

It is possible that the bucket is still full after the bucketized aggregation (if all of the slots store different keys). In this case, the new input row has to be inserted to the overflow bucket. This bucket is also aggregated by the bucketized aggregation procedure at the end of the processing. Nevertheless, the overflow happens infrequently in practice. In our evaluation, we found that the overflow did not happen at all when the hash table's occupancy was 50% or lower.

At the end of the processing, duplicate keys in each bucket need to be aggregated to obtain the final results. Although this can be done by invoking the bucketized aggregation procedure for every bucket in the hash table, Algorithm 3 can be slightly modified and vectorized to utilize SIMD lanes for processing multiple buckets simultaneously. If H_SIZE is the size of the hash table, the algorithm aggregates W buckets simultaneously, so there are H_SIZE/W outer loop iterations, and each iteration performs at most $W(W-1)/2$ comparisons. The overall time complexity of this algorithm is $O(H\_SIZE \cdot (W-1)/2)$. In our evaluation, this procedure entails less than 1% overhead of the entire processing.

Algorithm 4 Manage_Local_Hash_Table ($ltable$, $gtable$, $gbucket$, $ltog$, $gtol$, $touched$, $last\_remove$)

```
    ▷ Get the local buffer for the global bucket
 1: lbucket = gtol[gbucket]
    ▷ Case 1: there is a local buffer for the global bucket
 2: if (lbucket != -1) then
 3:     touched[lbucket] = 1
    ▷ Case 2: no local buffer, but there is empty local bucket
 4: else if (num_empty >= 0) then
 5:     lbucket = empty_buckets[num_empty − −]
        ▷ Build the mapping between local and global bucket
 6:     ltog[lbucket] = gbucket
 7:     gtol[gbucket] = lbucket
 8:     touched[lbucket] = 1
    ▷ Case 3: no local buffer or empty local bucket
 9: else
10:     for (i = 1; i <= L_SIZE; i++) do
            ▷ Search from the next of last spilled bucket
11:         lbucket = last_remove + i
12:         if (lbucket >= L_SIZE) then
13:             lbucket = lbucket - L_SIZE
14:         end if
            ▷ If the bucket is not used in current iteration
15:         if (touched[lbucket] == 0) then
16:             old_bucket = ltog[lbucket]
17:             gtol[old_bucket] = -1
18:             gtol[gbucket] = lbucket
19:             ltog[lbucket] = gbucket
20:             touched[lbucket] = 1
21:             Spill(ltable[lbucket], gtable[old_bucket])
22:             last_remove = lbucket
23:             break
24:         end if
25:     end for
26: end if
```

4 MANAGING HASH TABLE FOR MIMD (MANY-CORE) PARALLELIZATION

Today's processors combine SIMD parallelism on each core with multi/many cores. Thus, besides SIMD parallelization, we also need to be able to handle MIMD parallelism.

In the literature, three approaches have been explored for shared memory parallelization of hash table operations: use of atomic operations, use of locking, and use of separate hash tables [5]. Our target architecture in this paper, Intel Xeon Phi, does not support atomic writes of SIMD vectors, and thus, if different threads concurrently write SIMD vectors to same memory address, the actual value at that memory address will be non-deterministic. Combining SIMD execution and locking is also problematic, because a thread will need to lock the memory addresses on each of the SIMD lanes, thus, contention can really become a performance bottleneck.

Hence, using separate hash table for each thread is the most effective approach for combining SIMD and MIMD parallelism. The disadvantages of this approach, however, include: 1) merging the separate hash tables incurs overheads, and 2) memory requirements, especially for many-core systems, with the consequence that an input with a small group-by cardinality can reach the L2 cache barrier. In this section, we describe two techniques to optimize the performance of separate hash table, overcoming the two disadvantages, respectively.

4.1 Hybrid Hash Table

To enable better memory efficiency when the hash table is out of cache, we use a *hybrid hash table*, which allocates a small (cache resident) local hash table for each thread and synchronizes the local hash table with a larger global hash table when necessary. The hybrid hash table not only consumes less memory compared with separate hash table, but also reduces accesses to the shared memory if each thread has some data locality in its local hash table.

The hybrid hash table is especially suitable for combining SIMD with MIMD to accelerate our bucket hashing method because 1) SIMD does not support atomic operations and locking-based shared memory access is much less efficient than separate hash table, 2) bucket is a proper unit for maintaining the mapping from local to global hash table, and 3) scalar atomic instructions can be used for the synchronization of local hash table with global hash table with little interference in the SIMD processing of each thread.

Specifically, each thread first hashes a vector of keys to buckets in the global hash table. Instead of accessing the global hash table directly, the thread checks if these global buckets have buffers in the local hash table. If so, the computation is conducted on the local bucket. If no buffer is found, the thread checks if there are empty buckets in the local hash table. If so, a mapping between the local bucket and that global bucket is established, and the computation is conducted on the empty local bucket. If there is no empty bucket,

the thread needs to remove a bucket from the local hash table and spill it into the global hash table.

Algorithm 4 gives an implementation of this procedure. Given a bucket number in the global hash table *gbucket*, it returns the buffer in the local hash table *lbucket*. In the algorithm, the mapping between the local buckets and the global buckets is implemented by two index arrays, *ltog* and *gtol* – *ltog* maps from local to global, while *gtol* maps from global to local. The mapping is updated when an empty bucket is assigned to a global bucket or a local bucket is spilled into the global hash table. The `Spill` procedure uses scalar atomic instructions to achieve concurrent access to the shared hash table.

4.2 Parallel Reduction of Hash Tables

Merging two hash tables is more complex than merging scalars (or even arrays). Since the records need to be merged reside in the corresponding buckets, the `Reduction2` procedure, which reduces one hash table to another, is implemented by merging all of the buckets in the hash tables one by one. To merge a bucket ($B2$) to a corresponding bucket ($B1$), all of the records in $B2$ are checked to find a match or an empty slot in $B1$. If a match is found, the record in $B2$ is aggregated to the matching record in $B1$. On the other hand, if a match is not found, the record is stored in an empty slot in $B1$.

Suppose there are N threads participating in the computation, a straightforward way to reduce all of the hash tables is to invoke the `Reduction2` procedure $N-1$ times to reduce other hash tables to the main hash table one by one. Clearly, this is expensive. Our observation is that the tree-based reduction is applicable to hash tables. It takes $\lceil Log_2(N) \rceil$ steps to merge N hash tables. Initially, all of the hash tables are valid. In each step, half of the hash tables reduce themselves to the valid hash tables next them and invalidate themselves. The number of valid hash tables is reduced by half until there is only one hash table left, which stores the final results.

5 EVALUATION

In this section, we evaluate the efficiency of our vectorization methods. Our first set of experiments apply SQL-like queries on a set of synthetic dataset, chosen to cover different distributions and group cardinality values. Our second set of experiments compute analytical queries on `github datasets`[2].

In all experiments, we use multiplicative hashing as our hashing function, since it only involves multiplication and bit operations that are efficient for SIMD execution. Also, multiplicative hashing is *universal*, which means it is provably robust to adversarial distributions [5]. The constant multiplier is set to $(\sqrt{5}-1)/2$, which works reasonably well in practice [7].

Our experiments are conducted on an Intel Xeon Phi SE10P co-processor which has 61 cores (of 1.1GHz) with each core incorporated with 512-bit SIMD units. There are low-frequency and in-order x86 cores integrated on an Intel Xeon Phi, each of which supports as many as 4 hardware threads. Each cores has a local L1 cache of size 32KB, and a 512KB L2 cache which is interconnected coherently in a ring with other L2 cache (making the collective L2 cache size over 25MB). We use Intel ICC compiler 13.1.0 to compile all the codes with -O3 optimization enabled.

[2]https://www.githubarchive.org/

5.1 Database Queries on Synthetic Datasets

The first set of experiments computes answers to the following two queries, which are commonly used for evaluating aggregation performance [5, 6]. While an SQL-like syntax is used below, such queries arise from the computations in MapReduce or similar frameworks and in Statistical packages.

```
Q1: Select G, count(*), sum(V), sum(V*V)
    From R Group By G
Q2: Select G, max(V), min(V)
    From R Group By G
```

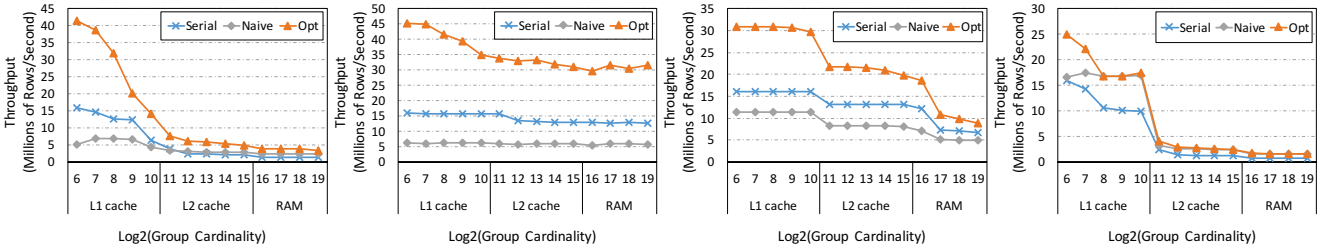
Here, `R` is a two-column table consisting of a group-by key `G` and an aggregation value `V`.

To cover different data characteristics, we generate datasets for `R` following different distributions and different group-by cardinality values, with each input containing 32M (2^{25}) rows. We mainly use 32-bit integer as the data type of both the key (`G`) and the value `V` as it is common in real-world aggregation tasks. We also conducted an experiment on 64-bit integer key with 64-bit double-precision value to show the performance variations with less SIMD lanes. Four distributions are used for our testing: heavy hitter (`HHitter`), `Zipf`, moving cluster (`MovCluster`), and `uniform`. The specific configurations for the four distributions are described in Section 3.1. The group cardinalities vary from 2^6 to 2^{19}. The size of hash table is 4KB (L1 cache resident) for datasets of group cardinality value between 2^6 and 2^{10}, and 64KB (L2 cache resident) for datasets of group cardinality value between 2^{11} and 2^{15}. When the group cardinality is larger than 2^{15}, the hash table is 1MB or more, i.e, out of cache.

5.1.1 Single-Core Performance. To evaluate single-core performance, we compare the throughput of three versions. The first version (`Serial`) is a serial implementation with a linear probing hash table. The second version (`Naive`) is an implementation of the straightforward SIMD aggregation as shown in Algorithm 2. The third version (`Opt`) is our optimized SIMD aggregation, which uses bucket hashing with offsets.

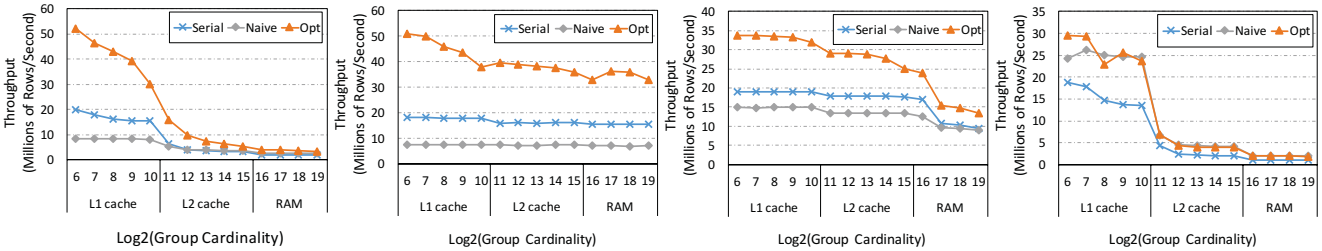
Figure 4 shows the throughput for `Q1` with inputs of different distributions and different group-by cardinality values. Recall that we had shown how the `Naive` version's performance is impacted by data conflicts in Section 3.1. These trends can be seen in the results here. For heavy hitter data, our SIMD approach has 2x to 2.7x throughput improvement over the serial approach, and up to 6x over the naive SIMD approach. The naive SIMD approach is even slower than the serial code when the hash table is in L1 cache because the overheads of SIMD operations outweigh the benefit of parallelization due to the data conflicts. The performance of all the three versions decreases as hash table becomes larger. The naive SIMD approach becomes faster than the serial code when the hash table is larger than L1 cache, but our approach still outperforms the naive approach by at least 1.6x and the serial code by at least 2x.

Note that hash-based aggregation involves intensive memory accesses in addition to hashing and computing. Take `Q1` for an example, to aggregate one row in `R`, at least six memory access operations are required, including three read operations (two for reading the key and the payload from `R`, and one for reading key in hash table) and three write operations (each for storing one of the three aggregation results). These memory accesses benefit little from SIMD instructions and only the hashing and computing part of the query can actually be accelerated by SIMD. Thus the SIMD speedup over the serial code is much less than the available lanes in a vector.



(a) HHitter (b) Zipf (c) MovCluster (d) Uniform

Figure 4: Throughput for Q1 with input of different distributions and different group cardinalities



(a) HHitter (b) Zipf (c) MovCluster (d) Uniform

Figure 5: Throughput for Q2 with input of different distributions and different group cardinalities

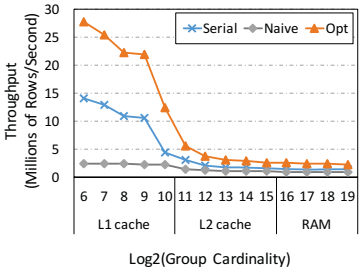


Figure 6: Throughput for Q1 on heavy hitter input of 64-bit data type

For Zipf data, the throughput of our SIMD approach is about 2.5x to 2.9x as much as the serial code, and 5x to 7x as much as the naive SIMD approach. The naive SIMD approach is again slower than the serial code because of the low SIMD utilization, arising from data conflicts. The performance of all the three versions is relatively steady as the hash table becomes larger.

For moving cluster data, the throughput of our SIMD approach is 1.6x to 2x as much as the serial code, and 1.8x to 2.7x as much as the naive SIMD approach. The performance of all three versions decreases when the hash table becomes larger than L1 cache and L2 cache, but our SIMD approach maintains a 1.6x speedup over the serial code with larger hash tables.

The naive SIMD approach is only effective on uniform data, accelerating the query with 2x throughput over the serial code as data conflicts are rare. Our SIMD approach has a similar throughput to the naive approach in this case, meaning it incurs little overhead for conflict mitigation on top of the SIMD processing. The performance of all three version decreases dramatically when the hash table becomes larger than L1 cache, but both the naive SIMD and our SIMD approach still outperform the serial code by at least 1.6x.

Figure 5 shows the throughput for `Q2` given the same input. The performance follows the same pattern as of `Q1`, except that its throughput is about 1.2x as much as `Q1` because there are only two aggregation operations in `Q2` instead of three.

The 512-bit SIMD vector can accommodate 16 lanes of 32-bit keys/values. To see the effectiveness of our approach with fewer SIMD lanes, we conduct an experiment on 64-bit data (64-bit integer key and 64-bit double-precision value), where only eight input rows can be processed simultaneously in the SIMD vector. The bucket size of our approach in this case is set to eight. Figure 6 shows the throughputs of Q1 on the 64-bit data of heavy hitter distribution. Our SIMD approach achieves 1.7x to 1.9x throughput improvement over the serial code, and has 2.5x to 11.5x speedups against the naive SIMD approach. The performance on other data distributions has a similar pattern to those in Figure 4, though the speedups for 64-bit data are not as high as for 32-bit due to fewer SIMD lanes.

Overall, we can achieve substantial speedups with SIMD parallelization. However, for the reasons explained above, they are not close to the width of the SIMD lanes. Yet, it is a significant improvement without the need for deploying additional hardware, compared to the options of not using SIMD parallelism, or using a naive approach.

5.1.2 Scaling across Cores. To evaluate the scalability of our approach across many/multi-cores, we test the performance of data aggregation using both our SIMD and MIMD parallelizing techniques. To illustrate the benefit from SIMD, we also implemented MIMD-only versions that use the same bucket hashing technique but without SIMD processing. We apply the separate table approach to hash tables in L1 and L2 cache and both the separate and the hybrid approach to hash tables in RAM.

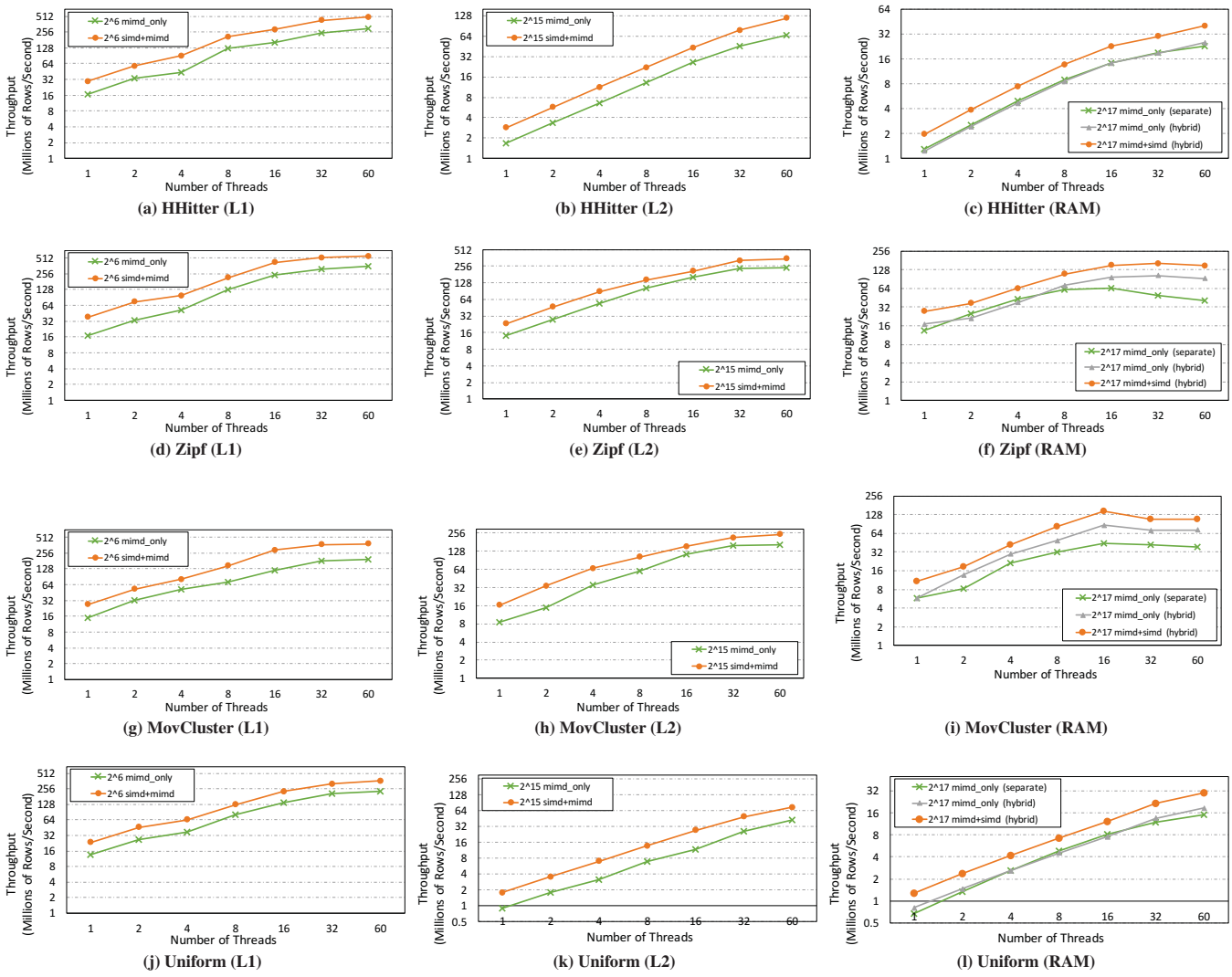


Figure 7: Scaling of Q1 across cores with hash tables of different sizes

Figure 7 shows the throughput for `Q1` on different number of cores and with different hash table sizes. Both x and y-axis in the figure use the base-2 logarithmic scale. For hash tables in L1 and L2 cache, the throughputs of both MIMD-only and SIMD+MIMD versions on all input datasets grow linearly as the number of cores increases. Compared with MIMD-only versions, SIMD+MIMD versions achieve an extra 1.6x to 2.5x speedups for the query. The speedups brought by SIMD are steady over 60 cores on most of the datasets, suggesting that SIMD benefits can be compounded with MIMD-only acceleration. Only in Figure 7e and Figure 7h, the benefits of SIMD shrink slightly when the number of cores is greater than 16 – this is because each core has less chance to reuse data in cache when the number of cores increases. This makes memory access operations more expensive in the query in relative to hashing and computing, and the latter is the part that can actually be accelerated by SIMD.

For hash tables in RAM, the throughputs of both MIMD-only and SIMD+MIMD versions grow as the number of cores increases in most cases, but the benefit of additional cores starts to decrease on some inputs with 32 or more cores. For Zipf and moving cluster data of 2^{17} group-by cardinality (Figures 7f and 7i), the throughputs of SIMD+MIMD reach maximum of 160 million records per second on 32 and 16 cores, and decrease with more than 32 cores. This is because, as the throughputs in these two cases are already high, enough work is not available for each core to justify the overheads associated with invoking more threads, synchronizing, and merging the local hash tables with the global table. The advantage of hybrid hash table also manifests on these two inputs especially with more cores, as the hybrid MIMD-only version achieves higher throughputs than the separate MIMD-only version. On the other hand, for heavy hitter and uniform data of 2^{17} group-by cardinality (Figures 7c and 7l), the throughputs continue to scale up to 60 cores. This is because each core has enough workload (as the throughputs are

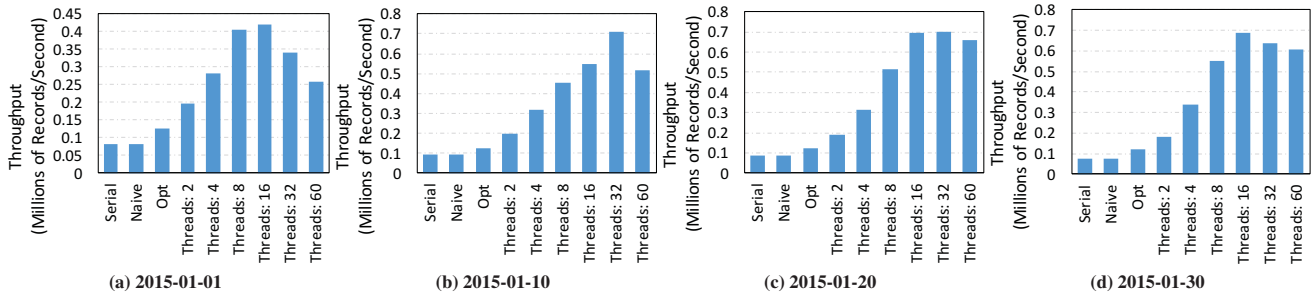


Figure 8: Throughput for different versions of Q3 with different inputs

relatively low) to justify the overhead. The relatively low throughputs also indicate that the data locality is not as good on these two inputs as on Zipf and moving cluster data. The hybrid approach does not show advantages over the separate approach on heavy hitter and uniform data, because the poor locality on each local hash table increases the swapping overhead between the local table and the global table.

The throughputs for `Q2` on different number of cores follow the same patterns as for `Q1`, and we omit the figures for `Q2` because of space limitations.

5.2 Analytic Queries on Real Datasets

The second set of experiments is conducted on dataset of repository operations on `github`. Each line in the dataset we obtained is a record in *json* format, which stores the information of a user event – including the event type (e.g., PUSH, CREATE, PULL), the actor of the event, and the repository the event is committed to, and so on. Given the records during a certain period of time, we answer the following query:

```
Q3: How many operations are committed for
    each type of event?
```

The inputs we use are the repository operation records on four dates, `2015-01-01`, `2015-01-10`, `2015-01-20`, and `2015-01-30`. The program first extracts the type of an event, and then, increments the count of that event.

We compare the throughput of a serial version (`Serial`), a naive SIMD version (`Naive`), and the version with our SIMD and MIMD method (`Opt`) running on different number of cores. Since the number of event types is always less than 100, a hash table of size 4 KB, which is L1 cache resident, is enough to host all the outputs.

Figure 8 shows the throughputs for the different versions of `Q3` with different inputs. With all of the four inputs, the naive SIMD version has slightly lower throughput than the serial version. The reason for the ineffectiveness of naive SIMD approach is that the conflict intensity values for `Q3` are high, ranging from 8.2 to 8.7. However, the difference of performance is not as significant as that for the database operations with heavy hitter input, which also has a conflict intensity of about 8. This is because the hashing operations, which are the actual parallel regions of `Q3`, account for a relatively smaller portion of the entire processing than those in `Q1` and `Q2`, as the parsing of the json format input is executed in sequential. Our SIMD approach achieves 1.4x - 1.6x speedup over the serial version. Over multiple cores, the throughput scales linearly when the number of threads is less than 8. The benefit of more cores starts to decrease when more threads are used, and the performance starts to drop when the number of threads reaches 32. The maximum throughput is achieved on 8 or 16 cores, with up to 9x speedup over the serial version. The scalability is expected to be better with larger input to justify the overhead of creating new threads.

6 RELATED WORK

We now compare our work with three areas that are close to our research, which are the efforts on parallelizing 'big-data' operations on multi-cores, optimizing them with SIMD features, and finally, the broader area of optimizing irregular applications with advanced SIMD instruction sets.

Optimizing Big-Data Operations on Multi-core CPUs Cieslewicz *et al.* [5] compared different ways of sharing hash table among the cores, and proposed an *adaptive* approach that chooses the sharing strategy based on the input data. They further proposed a general technique that detects and resolves data conflicts among the cores [6]. Their technique uses atomic operations to detect and measure data contention and clones the contentious data element once contention has been detected. Blanas *et al.* [2] argued that a simple join algorithm with shared hash table and no data partitioning can be very competitive to other (more complex) methods. Kim *et al.* [15] studied the effect of different optimizing strategies, including cache blocking and load balancing, for both hash and sort based join algorithms. They concluded that hash-based algorithm is more efficient when the output fits in the cache. The idea of bucket hashing with offsets to avoid data conflict is also similar to the idea of adopting different cache associativities to reduce cache-line conflicts [11].

Optimizing Big-Data Operations with Advanced SIMD An early effort to utilize SIMD on traditional CPUs for accelerating data analysis operations was by Zhou *et al.* [24]. Their implementation, however, is quite limited by the short SIMD vector and restricted SIMD operations. Polychronious *et al.* [20] used the early SIMD instructions to accelerate data aggregation with bucket hash table. In their approach, the SIMD vector is used for comparing a single input key with multiple keys in a bucket, and thus, only one input key is processed per SIMD instruction. They later refer this approach as *horizontal vectorization* [19]. More recent efforts have focused on newer instruction sets. Polychroniou *et al.* [19] proposed SIMD parallelization methods for most of the operations using advanced SIMD features such as*gather/scatter* instructions. They presented a new *vertical vectorization* approach which processes different input keys on different SIMD lanes. They also showed that the new approach is much more efficient than the horizontal vectorization. Jha *et al.* [12] utilize both SIMD and MIMD parallelism on an Intel Xeon Phi in improving the performance of hash joins. Their approach to vectorizing the hash table is also vertical, as multiple keys are hashed

and processed in a SIMD vector. Though both these efforts show gains from SIMD parallelization, their approaches are relatively inefficient in utilizing SIMD for hash tables, because of data conflicts. Polychroniou *et al.* detect conflicts among the SIMD lanes when building hash tables and only execute one of the conflicting lanes in a vector; their approach works well on *uniform* data in which the conflicts are rare. Jha *et al.* fall back on serial code when a SIMD vector needs to update hash tables.

These operations have also been studied on GPUs – where the execution follows the Single Instruction Multiple Thread (SIMT) parallelism. For our purposes, the key difference in SIMT and SIMD features is that the former supports atomic writes among threads. He *et al.* [9] exploited the SIMT feature of GPUs to improve the performance of hash joins. A co-processing technique for hash joins on the coupled CPU-GPU architecture is later proposed by He *et al.* [10].

Optimizing Irregular Applications with Advanced SIMD Because of conflicts that cannot be detected or resolved statically, hash-based aggregations are an example of irregular applications. There have only been a few attempts on mapping irregular applications to exploit SIMD parallelism. Saule and Catalyurek [21] provided a preliminary evaluation of graph applications on the the Xeon Phi architecture. Liu *et al.* [16] used ELLPACK sparse block format, to optimize SpMV kernel on the same architecture. Similarly, Tang *et al.* [23] utilized a hybrid storage format with jagged partitioning to optimize SpMV. More recently, Chen *et al.* [3, 13] proposed a tiling-and-grouping approach that improves the data locality and applies SIMD processing in a series of applications including graph algorithms, irregular reduction, molecular interation and matrix multiplication.

There have been several studies mapping irregular applications to SIMT processing on GPUs. For example, Merrill *et al.* [17] parallelized breadth-first search on the GPUs by focusing on fine-grained task management, and CuSha [14] optimizes graph processing on GPUs with intensive usage of shared memory and by re-organizing the graph data in shards.

7 CONCLUSION

This paper has presented a novel technique, bucket hashing with offsets, to improve SIMD parallelization of hash-based aggregation. By hashing keys from different SIMD lanes to distinct positions in the hash table and confining the probing in buckets, our method is able to significantly decrease data conflicts among lanes. The method also limits the overhead for aggregation within a bucket. An efficient bucketized aggregation procedure is invoked to either obtain final results, or to "squeeze" identical keys in a bucket to save space for new keys. We have also described two techniques, parallel reduction and hybrid hash table, to improve the performance of MIMD parallelization.

The experiments show that our vectorization method outperforms the naive SIMD parallelization. For skewed data, including heavy hitter, Zipf and moving cluster, our approach has 1.5x to 7x speedup over the naive SIMD approach, and achieves 1.6x to 2.9x speedup over the serial code on a single core of an Intel Xeon Phi. The throughput of our implementations increases linearly with increasing number of cores, until the overheads of merging separate hash tables and invoking new threads outweigh the benefits of parallelization.

Acknowledgements

This work was supported by NSF award CCF-1526386.

REFERENCES

[1] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-core, Main-memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (Sept. 2013).

[2] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*.

[3] Linchuan Chen, Peng Jiang, and Gagan Agrawal. 2016. Exploiting Recent SIMD Architectural Advances for Irregular Applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO 2016)*.

[4] Min Chen, Shiwen Mao, and Yunhao Liu. 2014. Big Data: A Survey. *Mobile Networks and Applications* (2014).

[5] John Cieslewicz and Kenneth A. Ross. 2007. Adaptive Aggregation on Chip Multiprocessors. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB '07)*.

[6] John Cieslewicz, Kenneth A. Ross, Kyoho Satsumi, and Yang Ye. 2010. Automatic Contention Detection and Amelioration for Data-intensive Operations. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*.

[7] Thomas H. Cormen, Clifford Stein, Ronald L. Rivest, and Charles E. Leiserson. 2001. *Introduction to Algorithms* (2nd ed.). McGraw-Hill Higher Education.

[8] Timothy Hayes, Oscar Palomar, Osman Unsal, Adrian Cristal, and Mateo Valero. 2016. Future Vector Microprocessor Extensions for Data Aggregations. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA '16)*.

[9] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational Joins on Graphics Processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*.

[10] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting Co-processing for Hash Joins on the Coupled CPU-GPU Architecture. *Proc. VLDB Endow.* 6, 10 (Aug. 2013).

[11] Changwan Hong, Wenlei Bao, Albert Cohen, Sriram Krishnamoorthy, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. 2016. Effective Padding of Multidimensional Arrays to Avoid Cache Conflict Misses. *SIGPLAN Not.* 51, 6 (June 2016).

[12] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. 2015. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *Proc. VLDB Endow.* 8, 6 (Feb. 2015).

[13] Peng Jiang, Linchuan Chen, and Gagan Agrawal. 2016. Reusing Data Reorganization for Efficient SIMD Parallelization of Adaptive Irregular Applications. In *Proceedings of the 2016 International Conference on Supercomputing (ICS '16)*.

[14] Farzad Khorasani, Keval Vora, Rajiv Gupta, and Laxmi N. Bhuyan. 2014. CuSha: Vertex-centric Graph Processing on GPUs. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*.

[15] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs. *Proc. VLDB Endow.* 2, 2 (Aug. 2009).

[16] Xing Liu, Mikhail Smelyanskiy, Edmond Chow, and Pradeep Dubey. 2013. Efficient Sparse Matrix-vector Multiplication on x86-based Many-core Processors. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing (ICS '13)*.

[17] Duane Merrill, Michael Garland, and Andrew Grimshaw. 2012. Scalable GPU Graph Traversal. *SIGPLAN Not.* 47, 8 (Feb. 2012).

[18] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-Efficient Aggregation: Hashing Is Sorting. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*.

[19] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*.

[20] Orestis Polychroniou and Kenneth A. Ross. 2013. High Throughput Heavy Hitter Aggregation for Modern SIMD Processors. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware (DaMoN '13)*.

[21] Erik Saule and Ümit V. Catalyurek. 2012. An Early Evaluation of the Scalability of Graph Algorithms on the Intel MIC Architecture. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW '12)*.

[22] Ambuj Shatdal and Jeffrey F. Naughton. 1995. Adaptive Parallel Aggregation Algorithms. *SIGMOD Rec.* 24, 2 (May 1995).

[23] Wai Teng Tang, Ruizhe Zhao, Mian Lu, Yun Liang, Huynh Phung Huynh, Xibai Li, and Rick Siow Mong Goh. 2015. Optimizing and Auto-tuning Scale-free Sparse Matrix-vector Multiplication on Intel Xeon Phi. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15)*.

[24] Jingren Zhou and Kenneth A. Ross. 2002. Implementing Database Operations Using SIMD Instructions. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*.