Accelerating Aggregation using Intra-cycle Parallelism

Ziqiang Feng, Eric Lo

Department of Computing
Hong Kong Polytechnic University
{cszqfeng, ericlo}@comp.polyu.edu.hk

***Abstract*—Modern CPUs have word width of 64 bits but real data values are usually represented using bits fewer than a CPU word. This underutilization of CPU at register level has motivated the recent development of *bit-parallel* algorithms that carry out data processing operations (e.g., filter scan) on CPU words packed with data values (e.g., 8 data values are packed into one 64-bit word). Bit-parallel algorithms fully unleash the intra-cycle parallelism of modern CPUs and they are especially attractive to main-memory column stores whose goal is to process data at the speed of the "bare metal". Main-memory column stores generally focus on analytical queries, where *aggregation* is a common operation. Current bit-parallel algorithms, however, have not covered aggregation yet. In this paper, we present a suite of bit-parallel algorithms to accelerate all standard aggregation operations: `SUM, MIN, MAX, AVG, MEDIAN, COUNT`. The algorithms are designed to fully leverage the intra-cycle parallelism in CPU cores when aggregating words of packed values. Experimental evaluation shows that our bit-parallel aggregation algorithms exhibit significant performance benefits compared with non-bit-parallel methods.**

I. Introduction

A *word* is the unit of data that a CPU core can process in one instruction. Modern CPUs have a typical *word width* of 64 bits, meaning the circuits in a processor core is able to simultaneously process 64 bits of information within one single clock cycle. This abundant *intra-cycle parallelism*, however, has been largely underutilized in data processing as most real data values could be represented using bits fewer than a CPU word. Therefore, when processing a, say, 7-bit attribute (e.g., an "age" attribute), normally each value is padded into a 64-bit word using 0's, wasting $64 - 7 = 57$ bits per cycle.

Recently, there are studies [1], [2] about *filter scan*, or simply *scan*, at the speed of the processing core. Their main idea is to fully utilize the intra-cycle parallelism in a CPU core when carrying out filter selection. Specifically, the techniques in [2] are especially designed for main-memory column stores [3], [4]. Multiple values (possibly encoded in compressed form [5], [6], [7]) from the same column are packed into words in memory and the words are organized into a layout that facilitates parallel predicate evaluation (filtering) by the CPU core[1]. Figure 1 shows such an example — eight values from the same column are packed into a 64-bit word in memory.

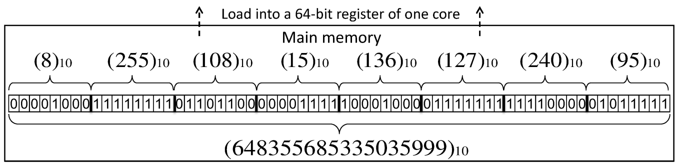

Fig. 1. Eight values are packed into a 64-bit word in memory.

When evaluating a predicate (e.g., price < 88), the eight values in the same word are evaluated in parallel within the same CPU register. Since the values are packed into a processor word, it is meaningless to directly evaluate the predicate on a *word of packed values* using ALU instructions. For example, in Figure 1, it is meaningless to compare whether the 64-bit word, whose plain value is $(648355685335035999)_{10}$, is less than 88. To carry out filter scan, a suite of *bit-parallel algorithms* is thus introduced in [2]. Each bit-parallel algorithm essentially turns a comparison operator ($=, \neq, <, >, \leq, \geq,$ BETWEEN) into a program of standard CPU instructions (e.g., logical AND ($\wedge$), exclusive OR ($\oplus$)) so that filter selections can be carried out on words of packed values directly.

Main-memory column stores generally focus on analytical queries, where aggregation is a common post-operation after filter scan. The bit-parallel algorithms in [2], however, do not cover aggregation. In this paper, we present a suite of efficient bit-parallel algorithms to implement all standard aggregation operations: `SUM, MIN, MAX, AVG, MEDIAN, COUNT`. The goal of the algorithms is to efficiently carry out bit-parallel aggregation on data filtered by a bit-parallel scan in a main-memory column store. Once again, the challenge is how to fully utilize the intra-cycle parallelism in CPU cores when aggregating words of packed values. Same as [2], our algorithms use no specialized but standard instruction sets found in all modern CPU architecture (including at the SIMD register level in most architecture): logical AND ($\wedge$), logical OR ($\vee$), exclusive OR ($\oplus$), binary addition ($+$), binary subtraction ($-$), negation ($\neg$), multiplication ($*$), and k-bit left or right shift ($\leftarrow_k$ or $\rightarrow_k$, respectively). Experimental evaluation shows that our bit-parallel aggregation algorithms exhibit significant performance gain compared with non-bit-parallel aggregation methods. Furthermore, our algorithms can integrate with SIMD instruction sets and multi-threading to enjoy further speed-up.

The remainder of this paper is organized as follows: Section II describes the preliminary and background information. Section III presents our bit-parallel algorithms for each aggregation function. Section IV contains our experimental

This work is partly supported by the Research Grants Council of Hong Kong (GRF PolyU 520413 and 521012) and a research gift from Microsoft Hong Kong.

[1]For example, the l_extendedprice attribute — the widest numeric attribute in TPC-H, can be encoded in 24 bits. So at least two values can be packed in a 64-bit CPU word.

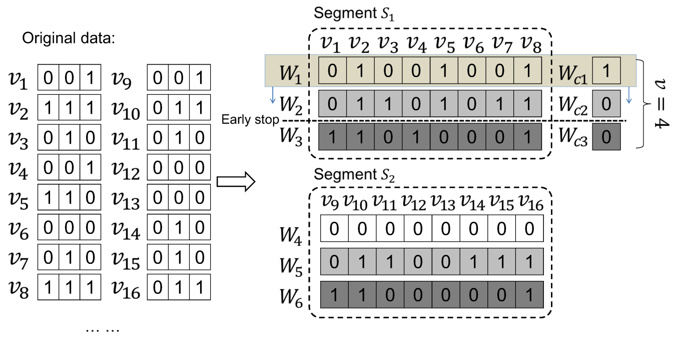

Fig. 2. VBP. A data value has $k = 3$ bits. A word has $w = 8$ bits.

TABLE I. NOTATION TABLE

Symbol	Meaning
W_i	the i-th word in a segment/sub-segment
F	filter bit vector
G_i	the i-th word-group
S_i	the i-th segment
SS_t	the t-th sub-segment
M	bit vector mask
v	a data value in the column
w	processor word width (bits)
n	number of tuples
k	number of bits to represent a value
τ	the size of a bit-group (bits)

results. Related work is covered in Section V. Finally, Section VI contains our concluding remarks.

II. Background and Preliminary

In this section, we give a brief introduction to the two bit-level main memory storage layouts and the bit-parallel scan algorithms proposed in [2]. We have developed two versions of bit-parallel aggregation algorithms, one for each storage layout. The first layout, VBP (Section II-A), uses a bit-level columnar data organization. The second layout, HBP (Section II-B), packs values from a column into processor words like Figure 1. The two layouts are designed for two different access patterns. For easy understanding, Table I summarizes the most frequently used symbols in this paper.

A. VBP: Vertical Bit Packing

The VBP storage format vertically packs the bits of a value across multiple words. Figure 2 shows the basic VBP layout under word width $w = 8$ bits and value width $k = 3$ bits. A fixed-length *segment* is formed for every k continuous word in the memory space. Each segment is a unit of data processed by a bit-parallel algorithm. In VBP, each segment holds column values from w tuples. Each value v stores its i-th bit in the i-th word. In Figure 2, eight values are packed into three 8-bit words, W_1, W_2, and W_3, in segment S_1.

During a predicate comparison, say, $v = 4$, the corresponding bit-parallel algorithm for that comparison operator first transposes the constant (i.e., value $100_2 = 4_{10}$) in the predicate and packs the bits vertically into k words (see words W_{c1}, W_{c2}, W_{c3} in Figure 2). Then the algorithm carries out predicate evaluation for a segment through k iterations. For the example in Figure 2, in the first iteration the algorithm compares the bits in word W_1 with W_{c1} and prunes values v_1, v_3, v_4, v_6, and v_7 because their most significant bits are 0's, which are different with W_{c1} — the most significant bit of $(\underline{1}00)_2$. The result of a comparison is stored in a *filter bit vector* F. The i-th bit in F indicates the result of applying the predicate on v_i. So, after this iteration, F is temporally $(\underline{0}1\underline{00}1\underline{00}1)_2$, with 0's indicating values v_1, v_3, v_4, v_6, and v_7 are not in the result. In the second iteration, the algorithm compares the bits in word W_2 with W_{c2} and further prunes values v_2, v_5, and v_8 because of the mismatch of the corresponding bits with the second bit of $(1\underline{0}0)_2$. The algorithm iterates through all k words in a segment, but it may stop early if all values are pruned. In the example, the algorithm jumps to process segment S_2 once the second iteration on segment S_1 has finished, because all values $v_1 \sim v_8$ have been pruned. Under VBP storage format, there would not be any overflow problem because the filter algorithms only use bitwise operations like AND, OR and XOR.

B. HBP: Horizontal Bit Packing

The HBP storage format horizontally packs values from a column into w-bit processor words. Figure 3a shows an example. Each k-bit value v_i is stored in a $(k+1)$-bit section whose leftmost bit is used as a delimiter between adjacent values of the same word. The delimiter bit is dedicated for handling overflow and producing the filter bit vector. A word can hold $\lfloor \frac{w}{k+1} \rfloor$ values. If w is not a multiple of $k+1$, 0's are right padded up to the word boundary. In HBP, a segment is formed for every $k+1$ contiguous processor words in memory space. In Figure 3a, eight values are packed into four 8-bit words in a segment. The values are packed in a "column-first" manner within a segment. For example, in segment S_1, v_1 is packed to W_1, v_2 is packed to W_2, but v_5 is packed to W_1. The reason of that is to facilitate the generation of the filter bit vector F.

During a predicate comparison, say, $v < 4$, the corresponding bit-parallel algorithm for that comparison operator first packs the constant in the predicate (i.e., value 4) repeatedly into a word (see word W_c in Figure 3b). Then, the algorithm applies a sequence of full-word instructions (e.g., exclusive OR $\oplus$, logical AND $\wedge$) to carry out the comparison to generate some intermediate results. Finally, a *filter bit vector* F is generated by shifting and applying logical OR to the intermediate results. In Figure 3b, the first (most significant) bit of the filter bit vector F indicates that v_1 is less than 4.

C. Cache Line Optimization

The basic format of VBP and HBP however may waste memory bandwidth if early stopping exists. Suppose that a CPU cache line contains 3 words. So, in Figure 2, it is possible that words W_1, W_2, and W_3 are in the same cache line. Skipping over W_3 that has already been loaded into the CPU cache with W_1 and W_2 results in wasted memory bandwidth. So, the actual storage format clusters words into the contiguous memory region called *word-group*.

Figure 4 shows an example of organizing eight 6-bit data values (i.e., $k = 6$) into two word-groups G_1 and G_2. The bits of value are split into *bit-groups* of size τ. In the example, $\tau = 3$, so, value $v_1 = \underline{110}\ \underline{011}$ is split into $\lceil \frac{k}{\tau} \rceil = 2$ bit-groups. The bit-groups of a value are packed to different words.

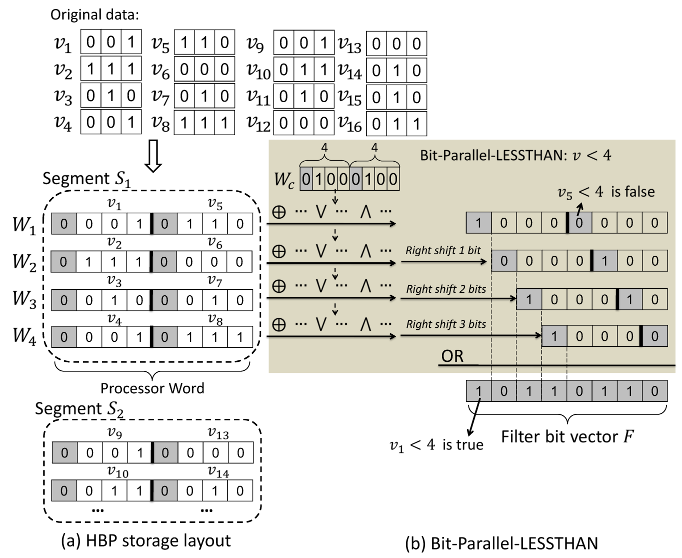

Fig. 3. HBP. A data value has $k = 3$ bits. A word has $w = 8$ bits.

For VBP (Figure 4a), the first bit-group of v_8 is vertically distributed to the last bit of words W_1, W_2, and W_3; and the second bit-group of v_8 is vertically distributed to the last bit of words W_4, W_5, and W_6. For HBP (Figure 4b), the first bit-group of v_1 is packed to word W_1 whereas the second bit-group of v_1 is packed to word W_2.

A word-group packs bit-groups into words of a continuous memory space. For example, in Figure 4a, word-group G_1 packs the first bit-group of all values together. So, during a filter scan, if we can prune all values after scanning $W_1 \sim W_3$ of segment S_1, we can skip the remaining words of S_1 and avoid bringing $W_4 \sim W_6$ into the cache. In this way the memory bandwidth could be used more judiciously.

In HBP, words that collectively contain all bits of a set of data values form a *sub-segment*. In Figure 4b, words W_1 and W_2 form a sub-segment SS_1 because they collectively contain all bits of values v_1 and v_5.

D. VBP versus HBP — A Remark

VBP and HBP are two orthogonal storage layouts. VBP is more space efficient because it does not require an extra delimiter bit for each value. However, VBP breaks a data value into more words, so when reconstructing a value back to its plain form, its performance is worse than HBP.

E. Evaluation of Filter Scan

Evaluations of complex predicates (e.g., predicates involving different columns from a table) in a filter scan are done by evaluating each simple predicate independently and combining their filter bit vectors. For example, the predicate `R.a>4 AND S.b = 10` can be evaluated by (i) applying `BIT-PARALLEL-GREATERTHAN` algorithm in [2] on column `R.a`, (ii) applying `BIT-PARALLEL-EQUAL` algorithm in [2] on column `S.b`, and (iii) intersecting their filter bit vectors.

III. Bit-Parallel Aggregation

The proposal in [2] has not covered any bit-parallel aggregation algorithms. To evaluate aggregation queries like

```
Q1: SELECT SUM(X) FROM Y WHERE Z < 4
```

it was suggested to first carry out a bit-parallel filter scan to obtain a filter bit vector F that indicates which tuples pass the filter. Then, a non-bit-parallel methodology is used: for each '1' in the filter bit vector F, its corresponding plain data value v is reconstructed from the horizontal/vertical bit-packed data and packed as a standalone 64-bit word. Afterwards, all those words go through CPU to carry out aggregation in plain form.

The above non-bit-parallel implementation of aggregation, however, has several drawbacks: (1) it exploits no intra-cycle parallelism at all; (2) even worse, it has to burn many instructions to reconstruct the data values back to their plain form. Using the HBP formatted data in Figure 3 as an example. Suppose that the filter bit vector F after a filter-scan `v < 4` is $(10110110)_2$ for segment S_1. The plain data value is reconstructed in four steps:

1) Identifies a value v from F that passes the filter: It starts with the rightmost 1 in F. To extract v, we compute its offset O in F by (i) $F \oplus -F$, (ii) apply the `POPCNT` bit-wise procedure[2] to count the number of 1's, (iii) add 1 to the count:

$$\begin{array}{lr} & F = (10110110)_2 \\ (i) & F \oplus -F = (11111100)_2 \\ (ii) & \texttt{POPCNT}(F \oplus -F) = (6)_{10} \\ (iii) & O = \texttt{POPCNT}(F \oplus -F) + 1 = (7)_{10} \end{array}$$

Through these instructions, value v_7 is the first value identified to be passing the filter from $F = (101101\underline{1}0)_2$.

2) Reconstruct the value of v by (i) modulo the offset O by the number of words in a segment to locate the word W_v that contains v; (ii) compute $\lfloor O/(k+1) \rfloor + 1$ to locate the position of v in W_v; (iii) right shift by $(\lfloor \frac{w}{k+1} \rfloor - \lceil O/(k+1) \rceil)(k+1)$ bits and mask the word W_v to make v as a full-word in plain:

$$O = \texttt{POPCNT}(F \oplus -F) + 1 = (7)_{10}$$

$$(i) \qquad 7 \texttt{ MOD } 4 = (3)_{10}$$

//found word W_3 is holding v_7

$$(ii) \qquad \lfloor 7/4 \rfloor + 1 = (2)_{10}$$

//found v_7 is the 2-nd value in W_3

$$(iii) \qquad \rightarrow_{(2-2)\times 4} W_3 \wedge (00000111)_2 = (00000\underline{010})_2$$

//v_7 in a full 8-bit word

3) Take away the reconstructed value v from F, by $F = F \wedge (F-1)$:

$$F = (101101\underline{1}0)_2$$
$$F := F \wedge (F-1) = (10110100)_2$$

4) Proceed to the next value in F that passes the filter: go to Step 1 with the updated F from Step 3. Stop when $F = 0$ (as Step 3 will unset the rightmost '1' of F each time).

[2]The `POPCNT` procedure stands for *population count*. It is a standard bit manipulation procedure. In our experiments, we use the `POPCNT` instruction provided by our Intel Haswell processor.

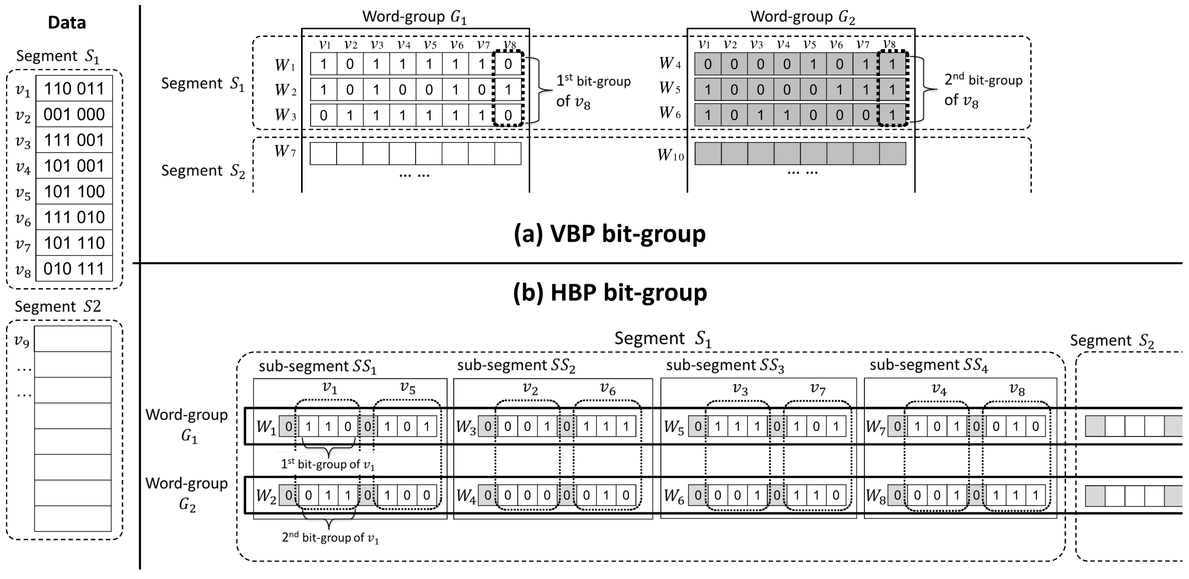

Fig. 4. Cache Optimization. A data value has $k = 6$ bits. A word has $w = 8$ bits. The size of a bit group is $\tau = 3$.

Aggregations are then either inlined within or carried out after the reconstruction process. For example, SUM aggregation can be done by inlining a running sum to accumulated the values reconstructed at Step 2(iii).

From the above discussion, we see how non-bit-parallel aggregations burn instructions by reconstructing the plain data values. The overheads in VBP are even higher because the bits of a value are distributed into more words, which further increases the effort to reconstruct the values back to their plain form. Of course, those overheads are negligible when the filter is highly selective. However, many realistic analytic queries are indeed not highly selective. For example, we found that almost half (10 out of 22) of the TPC-H queries have selectivity of 0.01 or above. For those queries, we observe that, if filter scans are carried out using the bit-parallel algorithms in [2], then the non-bit parallel implementation of aggregation takes from 30% (Q6: 0.6/2.1) to 99% (Q1: 85.6/86) of the whole query execution (see the shaded cells in Table II).

In the next section, we present our bit-parallel algorithms to implement various aggregate operators under the VBP and HBP storage layouts. Our algorithms take as input a filter bit vector returned by a bit-parallel scan operator and compute the aggregate without reconstructing the values back to their plain form. We regard our algorithms as additional access methods for the optimizer to consider when the queries are not highly selective. Those queries deserve attention not only because they are common in real workloads (e.g., 10 out of 22 TPC-H queries), but also because of their generally longer execution time — obviously they are the ones that zoom the bad side of the database product and being able to significantly reduce their running times would be much more important than reducing the running times of the already fast running ones. We remark that our bit-parallel solutions are orthogonal to solutions using specialized hardware such as GPU [8], [9]. That is, our algorithms can be applied at the register level of GPU. Our discussion assumes the aggregations work on unsigned integers[3]. NULL values can be handled using the techniques in [10]. To accelerate analytical processing, recent work suggests denormalizing the database (e.g., pre-joining all tables as a *wide table*) to eliminate join processing and materializing extra columns (e.g., an attribute in a different sort order as a column, expression results over multiple attributes as a column) to facilitate grouping, complex expression computation, and selection [11], [12]. We follow those approaches in this paper. Therefore, operations such as joins and group-by could be transformed into simple scans, and aggregations could directly work on a single column. Such approach has been evaluated to be practical and efficient in main memory analytical databases while the extra space requirement is not an issue because of the use of compressions in columnar [11].

A. Bit-Parallel Aggregation Under VBP

In this section, we present the bit-parallel algorithms for implementing COUNT, SUM, MIN, MAX, MEDIAN under the VBP storage format. The AVG aggregation is simply implemented by dividing the sum with the count.

[COUNT] Computing the COUNT aggregation is straightforward — we simply add up the number of 1's in the filter bit vector F for each segment S. In terms of implementation, we use the POPCNT procedure to count the number of 1's in a bit vector. Then, the overall count aggregate is computed by:

$$\sum_{\text{segment } S} \texttt{POPCNT}(S.F)$$

Let n be the number of values. Since a segment generally contains w values, the number of segments is given by $\frac{n}{w}$. Hence the complexity of COUNT is $O(\frac{n}{w})$.

[3]Other numeric types like signed integers and floating point with limited precision can be mapped to unsigned integers with a scaling scheme [7].

[SUM] Notice that a value v of k bits can be expressed as $v = \sum_{j=1}^{k} v^{(j)} \times 2^{k-j}$, where $v^{(j)}$ means the j-th bit of v (the 1-st bit is the most significant bit). Take values v_1 and v_2 in Figure 2 as an example:

$$\begin{aligned} v_1 &= 001_2 = 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\ v_2 &= 111_2 = 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 \end{aligned}$$

Hence, the sum of w values can be computed in the form:

$$\begin{aligned} \sum_{i=1}^{w} v_i &= \sum_{i=1}^{w}\sum_{j=1}^{k} v_i^{(j)} \times 2^{k-j} \\ &= \sum_{j=1}^{k}\left(\sum_{i=1}^{w} v_i^{(j)}\right) \times 2^{k-j} \end{aligned}$$

The sum of Segment S_1 in Figure 2 can then be computed through (bits of v_1 are especially underlined to ease understanding):

$$\begin{aligned} & v_1 + v_2 + v_3 + \ldots + v_8 \\ =\ & \underline{001} + 111 + 010 + \ldots + 111 \\ =\ & (\underline{0} \times 2^2 + \underline{0} \times 2^1 + \underline{1} \times 2^0) \\ & + (1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0) \\ & + (0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0) \\ & + \ldots \\ & + (1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0) \\ =\ & (\underline{0} + 1 + 0 + \ldots + 1) \times 2^2 \text{———}(*) \\ & + (\underline{0} + 1 + 1 + \ldots + 1) \times 2^1 \\ & + (\underline{1} + 1 + 0 + \ldots + 1) \times 2^0 \end{aligned}$$

Especially, line (*) above is essentially counting the number of 1's in word W_1 and multiplying that count with 2^2. The latter can be implemented with 2-bit left shift instruction. Therefore, computing the sum over a segment in a bit-parallel fashion can be done by $\sum_{i=1}^{k} \leftarrow_{k-i}$ POPCNT(W_i). For segment S_1, that is:

$$\begin{aligned} & v_1 + v_2 + v_3 + \ldots + v_8 \\ =\ & \leftarrow_2 \texttt{POPCNT}(W_1) \\ & + \leftarrow_1 \texttt{POPCNT}(W_2) \\ & + \leftarrow_0 \texttt{POPCNT}(W_3) \\ =\ & \leftarrow_2 (3) + \leftarrow_1 (5) + \leftarrow_0 (4) \\ =\ & 26 \end{aligned}$$

Algorithm 1 shows our implementation of SUM. We keep a temporary array $bSum[1 \ldots k]$ to accumulate the POPCNT(W_i) across the segments. We process the data in one word-group after another. For each word-group G, we iterate over each segment S, and add the POPCNT of each word $S.W_i$ to the corresponding $bSum[i]$ entry. For example, when processing segment S_2 in Figure 4a, the POPCNT of W_7 (the $\underline{1}$-st word of S_2) is added to $bSum[\underline{1}]$. To compute the SUM over the values that pass through a previous filter, we intersect W_i with the filter bit vector F before the counting and shifting (Line 5). By having the temporary array $bSum$, we can carry out the shifts in the end (Line 6), reducing the number of shifts to k in total instead of k per segment. The algorithm logically scans every word in the data exactly once. The complexity of SUM is therefore $O(\frac{nk}{w})$.

Algorithm 1 Bit-Parallel SUM in VBP

```
1: Initialize $bSum[1 \ldots k] := 0$: bit-wise sums
2: for each word-group $G$ do
3:     for each segment $S$ in column $C$ do
4:         for each $S.W_i$ in $G$ do
5:             $bSum[i] := bSum[i] + \texttt{POPCNT}(S.W_i \wedge S.F)$
6: $sum = \sum_{i=1}^{k} \leftarrow_{k-i} bSum[i]$
7: return $sum$
```

[MIN] Each segment in VBP contains w data values. MIN aggregation relies on a bit-parallel procedure SLOTMIN (stands for slot-wise minimum) that compares the w data values in a segment S_x with another w data values in a segment S_y in a slot-wise fashion and returns the minimum value of each slot. Using the values in segment S_1 and segment S_2 in Figure 2 as an example, on plain value level, the SLOTMIN procedure produces the result as follows:

$$\begin{aligned} S_1 &= \{1,7,2,1,6,0,2,7\} \\ S_2 &= \{1,3,2,0,0,2,2,3\} \\ \texttt{SLOTMIN}(S_1,S_2) &= \{1,3,2,0,0,0,2,3\} \end{aligned}$$

That is, the i-th value in the result of a SLOTMIN procedure is the minimum between the two i-th values in both segments. With the SLOTMIN procedure, we can maintain a temporary segment S_{temp} and iterate through each segment S_x to update $S_{\text{temp}} :=$ SLOTMIN(S_x, S_{temp}). The final S_{temp} would then contain the w smallest numbers across all segments.

The bit-parallel implementation of SLOTMIN(S_x, S_y) procedure requires the computation of a w-bit vector M_{lt} whose i-th bit is 1 if the i-th value in S_x is less than the i-th value in S_y or 0 otherwise. For example, the bit vector M_{lt} between S_1 and S_2 is $(00000100)_2$, meaning the 6-th slot of S_1 (i.e,. value 0) is less than that of S_2 (i.e,. value 2). By having bit vector M_{lt}, we can use that to pick the smaller value between S_1 and S_2 on each slot by:

$$(M_{lt} \wedge S_x.W_i) \vee (\neg M_{lt} \wedge S_y.W_i)$$

So, for S_1 and S_2 in Figure 2, on bit-level, SLOTMIN(S_1, S_2) works as follows:

			SLOTMIN(S_1, S_2)
	S_1	S_2	$(M_{lt} \wedge S_1.W_i) \vee (\neg M_{lt} \wedge S_2.W_i)$
W_1	0100 1001	0000 0000	0$\underline{0}$00 0000
W_2	0110 1011	0110 0111	0$\underline{1}$10 0011
W_3	1101 0001	1100 0001	1$\underline{1}$00 0001

The result of the above bit-parallel SLOTMIN procedure, of course, is interpreted vertically under VBP. For example, the second bit of each word in SLOTMIN (underlined above) collectively means $(011)_2$, i.e., among the second slots in S_1 and S_2, the value $(011)_2 = 3_{10}$ is the smaller one.

Algorithm 2 shows our implementation of MIN aggregation. We iterate through all segments (Lines 2–3) to obtain the overall slot-wise minimum. The slot-wise minimum contains the w smallest numbers across all segments. Next, we simply reconstruct these w values to plain form and return the minimum one (Lines 4–6). Note that this step only reconstructs a small constant number (i.e., w) of values, we found the cost of this step is negligible. The overall cost of the algorithm is therefore dominated by Lines 2–3, given by $O(\frac{nk}{w})$.

In SLOTMIN, we use the BIT-PARALLEL-LESSTHAN algorithm in [2] to obtain the bit vector M_{lt} (Line 9). Additionally, to discard values that do not pass the filter F, we intersect M_{lt} with F before finding the slot-min (Line 10).

Algorithm 2 Bit-Parallel MIN in VBP

```
1: Initialize $S_{temp}$: a running slot-wise minimum segment
2: for each segment $S$ in column $C$ do
3:     $S_{temp} := \text{SLOTMIN}(S, S_{temp}, S.F)$
4: Reconstruct the values in $S_{temp}$ to plain form
5: $min :=$ the minimum among the $w$ reconstructed values
6: return $min$

7: procedure SLOTMIN($S_x$, $S_y$, $F$)
8:     Initialize $S_{min}$
9:     $M_{lt} := \text{BIT-PARALLEL-LESSTHAN}(S_x, S_y)$
10:    $M_{lt} := M_{lt} \wedge F$
11:    for each word-group $G$ do
12:        for each word $W_i$ of $S_x, S_y$ in $G$ do
13:            $S_{min}.W_i := (M_{lt} \wedge S_x.W_i) \vee (\neg M_{lt} \wedge S_y.W_i)$
14:    return $S_{min}$
```

[MAX] MAX aggregation is implemented like MIN aggregation. The worth noticing changes just include: (1) we need a SLOTMAX procedure to compute the slot-wise maximum, and (2) we leverage the BIT-PARALLEL-GREATERTHAN procedure in [2] to obtain the bit vector M_{gt}.

[MEDIAN] We begin the explanation of computing MEDIAN by taking segment S_1 in Figure 2 as an example. Since segment S_1 has an even number of values, without loss of generality, we look for the lower median, i.e., the 4-th smallest value in the segment.

Our idea is to progressively determine the value of the median bit by bit, from its most significant bit to its least significant bit. The median $\mathcal{M}$ should be a k-bit value. So, we denote the i-th bit of the median as $\mathcal{M}[i]$. To determine the 1-st bit of the median, i.e., $\mathcal{M}[1]$, we count the number of 1's in W_1:

$$\text{POPCNT}(W_1) = 3$$

This result translates into the fact that there are only 3 values in segment S_1 greater than $(100)_2 = (4)_{10}$. As we are currently looking for the 4-th smallest value in the segment, we assert that (i) the median must be smaller than 4 and (ii) values with 1 in their first bit in W_1, i.e., v_2, v_5, and v_8, are not the answer. Furthermore, by (i) we establish the first bit of the median as 0, i.e., $\mathcal{M}[1] = 0$, so as to ensure it has a value smaller than 4.

Next, we proceed to determine the second bit of the median, i.e., $\mathcal{M}[2]$, by working on w_2. As such, we count the number of 1's in $W_2 \wedge \neg W_1$:

$$\text{POPCNT}(W_2 \wedge \neg W_1) = 2$$

This result translates into the fact that there are only 2 values in segment S_1 greater than $(010)_2 = (2)_{10}$, despite values v_2, and v_5, and v_8 (by $\wedge\neg W_1$) . As we are currently looking for the 4-th smallest value in the segment, we assert that (i) the median should have a value in $[2, 4)$, and (ii) values with 1 in the second bit in $\neg W_1 \wedge W_2$, i.e., v_3 and v_7, are the median candidates. So by (i), we can assert $\mathcal{M}[2] = 1$.

Finally, we proceed to determine the last bit of the median, i.e., $\mathcal{M}[3]$, by working on W_3. As such, we count the number of 1's in $W_3 \wedge W_2 \wedge \neg W_1$:

$$\text{POPCNT}(W_3 \wedge W_2 \wedge \neg W_1) = 0$$

This result translates into the fact that both candidate values v_3 and v_7 have 0's in their last bit. It also tells us that the last bit of the median must be 0, i.e., $\mathcal{M}[2] = 0$. So, all together the median is $(010)_2$.

Algorithm 3 Bit-Parallel MEDIAN in VBP

```
1: Initialize $u := \text{COUNT}(*)$: the number of candidates
2: Initialize $r := \lfloor u/2 \rfloor$: the rank of (lower) median
3: Initialize $\mathcal{M} := 0$: the running result of median
4: for each segment $S$ in column $C$ do
5:     $S.V := S.F$: the candidate bit vector
6: for each word-group $G$ do
7:     for each word $W_i$ of all segments do
8:         $c := \sum_{\text{segment } S:S.V \neq 0} \text{POPCNT}(S.V \wedge S.W_i)$
9:         if $u - c < r$ then                ▷ the $i$-th bit must be 1.
10:            $\mathcal{M}[i] := 1$
11:            $r := r - (u - c)$
12:            $u := c$
13:            for each segment $s$ in column $C$ do
14:                $s.V := s.V \wedge s.W_i$
15:        else                               ▷ the $i$-th bit must be 0.
16:            $\mathcal{M}[i] := 0$
17:            $u := u - c$
18:            for each segment $s$ in column $C$ do
19:                $s.V := s.V \wedge \neg s.W_i$
20: return $\mathcal{M}$
```

Algorithm 3 shows our implementation of MEDIAN. We initialize a candidate bit vector V to each segment as F to hold the median candidates (Lines 4–5). Then we iterate through the word-groups. Within each word-group, we process the first word in each segment in order to determine the first bit of $\mathcal{M}$, and then we process the second word in each segment, and so on. In this way, we examine the i-th bits of all data for $i := 1 \ldots k$. During each iteration, after we have determined the current i-th bit of the median is 1 (Line 9) or 0 (line 15), we update the candidate set V. Accordingly, we also update the number of candidates (u) and the target's rank (r) in the candidate set. Since the candidate bit vector $S.V$ becomes more and more sparse (i.e., containing fewer 1's) when the algorithm proceeds, we can possibly skip POPCNTing a whole segment by testing $S.V \neq 0$ (Line 8), thus saving some POPCNT instructions. Note that Algorithm 3 essentially solves any r-selection problem, i.e., finding the r-th smallest value in a list, by controlling the value of r. The algorithm complexity is similar to SUM in that it asymptotically scans each data word once. The complexity of the algorithm is $O(\frac{nk}{w})$.

B. Bit-Parallel Aggregation Under HBP

In this section, we present the bit-parallel algorithms for implementing aggregation under the HBP storage format. We note that the implementation of COUNT under HBP is identical to the implementation under VBP because COUNT aggregation simply counts the number of 1's in the filter bit vector. Similar to VBP, the implementation of MAX here is a straightforward adaption of MIN, so we omit it for brevity. Of course, the AVG aggregation is still implemented by dividing the sum with the count. So, in the following, we only focus on SUM, MIN, and MEDIAN.

[SUM] Notice a value in HBP is split and distributed into $B = \lceil \frac{k}{\tau} \rceil$ bit-groups. To reconstruct a value, we can carry out:

$$v = \sum_{g=1}^{B} \leftarrow_{(B-g)\tau} (v.b_g)$$

Here, $v.b_g$ means the g-th bit-group of v. For instance, in Figure 4b, we have $v_1.b_1 = (110)_2, v_1.b_2 = (011)_2$ and $v_1 = \leftarrow_3 (110_2) + \leftarrow_0 (011_2)$.

Then, the sum over all values is:

$$\sum_i v_i = \sum_i \sum_{g=1}^{B} \leftarrow_{(B-g)\tau} (v_i.b_g)$$
$$= \sum_{g=1}^{B} \leftarrow_{(B-g)\tau} (\sum_i v_i.b_g)$$

Note that $\sum_i v_i.b_g$ is essentially the sums of the *in-word-sum* of all words in the g-th word-group. Consider the word-group G_1 in Figure 4b as an example. The in-word-sum of word W_1 is $(110)_2 + (101)_2$. Suppose there is a procedure IN-WORD-SUM for computing the in-word-sum of a word, then for word-group G_1, $\sum_i v_i.b_1$ equals to $\sum_{i=1,3,5,7,\ldots}$ IN-WORD-SUM(W_i).

We now present the details of implementing the IN--WORD-SUM procedure. Our proposed algorithm is inspired by the Gilles-Miller method for sideways addition [13]. To illustrate, we use a 32-bit word W as an example ($w = 32$, $k = 6$, $\tau = 3$):

$$W = 0\,\underbrace{000}_{0}\,0\,\underbrace{001}_{1}\,0\,\underbrace{010}_{2}\,0\,\underbrace{011}_{3}\,0\,\underbrace{100}_{4}\,0\,\underbrace{101}_{5}\,0\,\underbrace{110}_{6}\,0\,\underbrace{111}_{7}$$

So, for W, its in-word-sum is $0 + 1 + ... + 7 = 28$. To compute the in-word-sum from W, we have to carry out four instructions:

1) Compute the pair-wise sum on W by $W := W + \rightarrow_{\tau+1} (W)$:

$$W := W + \rightarrow_4 (W) =$$
$$\underbrace{0000}_{0}\ \underbrace{0001}_{0+1}\ \underbrace{0011}_{1+2}\ \underbrace{0101}_{2+3}\ \underbrace{0111}_{3+4}\ \underbrace{1001}_{4+5}\ \underbrace{1011}_{5+6}\ \underbrace{1101}_{6+7}$$

2) Remove the redundant pair-wise sums by $W := W \wedge 0^{\tau+1}1^{\tau+1} \ldots 0^{\tau+1}1^{\tau+1}$ (Note that we use exponentiation to denote bit repetition, e.g., $1^5 0^2 = 1111100, 001^k = 00\underbrace{11\cdots11}_{k}$):

$$W := W \wedge 0^4 1^4 0^4 1^4 \ldots 0^4 1^4 =$$
$$0000\ \underbrace{0001}_{0+1}\ 0000\ \underbrace{0101}_{2+3}\ 0000\ \underbrace{1001}_{4+5}\ 0000\ \underbrace{1101}_{6+7}$$

3) Add each pair-wise sum to the sums on its left by $W := W * 0^{2\tau+1}1 \cdots 0^{2\tau+1}1$:

$$W := W * 0^7 1 \cdots 0^7 1 =$$
$$\underbrace{0001\ 1100}_{0+1+...+6+7}\ \underbrace{0001\ 1011}_{2+3+...+6+7}\ \underbrace{0001\ 0110}_{4+5+6+7}\ \underbrace{0000\ 1101}_{6+7}$$

4) Right shift by $w - 2(\tau+1)$ bits to get the answer in place:

$$W := \rightarrow_{24} (W) =$$
$$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ \underbrace{0001\ 1100}_{0+1+...+6+7}$$

Algorithm 4 shows our implementation of SUM in HBP. We iterate through all sub-segments (Lines 2–6) to accumulate the *in-word-sum* for each word-group. $SS_t.W_i$ in Line 6 means the word of sub-segment SS_t that falls in word-group G_i. As we need to take into account the filter result F, we use the procedure GET-VALUE-FILTER(SS_t, F) to obtain a filter mask M for that specific sub-segment before computing its in-word-sum (Line 4).

Specifically, the procedure GET-VALUE-FILTER(SS_t, F) returns a value filter for the t-th sub-segment in segment S (Lines 15–18). That value filter could be used to wipe out values in a word that are not in the filter. Using Figure 4b as an example again. Suppose we have a filter bit vector $F = 11\mathbf{0}1\ 01\mathbf{1}0$, indicating that v_7 passes the filter but v_3 does not, in sub-segment SS_3. Therefore, when working on word $W_5 = 0\,\underbrace{111}_{v_3.b_1}\ 0\,\underbrace{101}_{v_7.b_1}$ in sub-segment SS_3, GET-VALUE-FILTER(SS_t, F) carries out the following:

1) Prepare a delimiter filter M_d, where the delimiter bits indicate which values in that sub-segment pass the filter. It works by $(\leftarrow_{t-1} F) \wedge 10^\tau \cdots 10^\tau$ (Line 16). For word W_5 in sub-segment SS_3 in Figure 4b:

$$M_d := (\leftarrow_2 F) \wedge \mathbf{1}000\ \mathbf{1}000 = \mathbf{0}000\ \mathbf{1}000$$

2) Now, the delimiters in M_d show that v_7 passes the filter but v_3 does not. Next, we create a mask M to extract the values that passes the filter by $M := M_d - \rightarrow_\tau (M_d)$ (Line 17). So, we get:

$$M := M_d - \rightarrow_3 (M_d) = 0\mathbf{000}\ 0\mathbf{111}$$

By having that filter M, when carrying out the in-word-sum (Line 6), we apply the filter M to W_5:

$$W_5 \wedge M = 0\ \underline{000}\ 0\,\underbrace{101}_{v_7.b_1}$$

In the above, we see the value of v_3 is wiped out from word W_5 and would not contributed to the sum. Lastly, we add the the bit-group-wise sums together with proper shifts (Line 7) to obtain the final result. As HBP storage requires extra delimiter bits, the number of sub-segment per segment is $\tau+1$, while the number of bit-groups is $\frac{k}{\tau}$. The complexity of the algorithm is slightly higher than that of VBP: $O(\frac{nk(\tau+1)}{w\tau})$.

Algorithm 4 Bit-Parallel SUM in HBP

```
1: Initialize $G_i.sum := 0, i = 1 \ldots B$: partial sum for each bit-group
2: for each segment $S$ in column $C$ do
3:     for each sub-segment $SS_t$ in $S$ do
4:         $M :=$ GET-VALUE-FILTER($SS_t, S.F$)
5:         for each word-group $G_i$ do
6:             $G_i.sum := G_i.sum +$ IN-WORD-SUM($SS_t.W_i \wedge M$)
7: $sum := \sum_{i=1}^{B} \leftarrow_{(B-i)\tau} G_i.sum$
8: return $sum$
9: procedure IN-WORD-SUM($W$)
10:     $W := W + \rightarrow_{\tau+1} (v)$
11:     $W := W \wedge 0^{\tau+1}1^{\tau+1}0^{\tau+1}1^{\tau+1} \ldots 0^{\tau+1}1^{\tau+1}$
12:     $W := W * 0^{2\tau+1}10^{2\tau+1}1 \ldots 0^{2\tau+1}1$
13:     $W := \rightarrow_{w-2(\tau+1)} (W)$
14:     return $W$
15: procedure GET-VALUE-FILTER($SS_t$, $F$)
16:     $M_d := (\leftarrow_{t-1} F) \wedge 10^{\tau}10^{\tau} \ldots 10^{\tau}$
17:     $M := M_d - \rightarrow_{\tau} (M_d)$
18:     return $M$
```

[MIN] MIN aggregation on HBP can be carried out in a way similar to MIN aggregation on VBP (Section III-A). Specifically, it relies on a bit-parallel procedure SUB-SLOTMIN that computes the slot-wise minimum between two *sub-segments*. Using sub-segment 1 (SS_1) and sub-segment 2 (SS_2) in Figure 4b as example, SUB-SLOTMIN(SS_1, SS_2) is:

$$SS_1 = \left\{\begin{matrix} 0\ 110 & 0\ 101 \\ \underbrace{\underline{0\ 011}}_{v_1=51} & \underbrace{\underline{0\ 100}}_{v_5=44} \end{matrix}\right\}$$

$$SS_2 = \left\{\begin{matrix} 0\ 001 & 0\ 111 \\ \underbrace{\underline{0\ 000}}_{v_2=8} & \underbrace{\underline{0\ 010}}_{v_6=58} \end{matrix}\right\}$$

$$\texttt{SUB-SLOTMIN}(SS_1, SS_2) = \left\{\begin{matrix} 0\ 001 & 0\ 101 \\ \underbrace{\underline{0\ 000}}_{8} & \underbrace{\underline{0\ 100}}_{44} \end{matrix}\right\}$$

To find the global minimum, we maintain a temporary sub-segment SS_{temp} and iterate through each sub-segment SS_x to update $SS_{temp} :=$ SUB-SLOTMIN(SS_x, SS_{temp}). After that, SS_{temp} will have $\frac{w}{\tau+1}$ values and the final answer can be easily found by reconstructing those values back to the plain form. For $w = 64$, $\tau = 3$, this last step only reconstructs 16 values back to their plain form, so the overhead is negligible.

The implementation of SUB-SLOTMIN procedure is similar to the implementation of SLOTMIN in VBP (Section III-A). It also requires the computation of a w-bit vector M_{lt} whose the i-th delimiter bit in M_{lt} is 1 if the i-th slot of SS_1 is less-than the i-th slot of SS_2. For the example above, M_{lt} of SUB-SLOTMIN(SS_1, SS_2) is **0**000 **1**000, meaning the first slot of SS_1 (i.e., 51) is not less than that of SS_2 (i.e., 8) while the second slot of SS_1 (i.e., 44) is less than that of SS_2 (i.e., 58). Moreover, we need to transform M_{lt} into a mask M so as to extract the slot-wise minimum from the two sub-segments. So, on bit-level, SUB-SLOTMIN(SS_1, SS_2) works as follows:

			SUB-SLOTMIN(SS_1, SS_2)
	SS_1	SS_2	$(M \wedge SS_1.W_i) \vee (\neg M \wedge SS_2.W_i)$
W_1	0110 0101	0001 0111	0$\underline{001}$ 0101
W_2	0011 0100	0000 0010	0$\underline{000}$ 0100

Algorithm 5 shows our implementation of MIN in HBP. We iterate through all sub-segments to obtain the overall slot-wise minimum in SS_{temp} (Lines 2–5). To take into account the filter bit vector F, for each sub-segment SS_t, we again prepare a delimiter filter M_d (Line 4) and pass that together with two sub-segments to the procedure SUB-SLOTMIN. Inside the procedure, we intersect M_d with M_{lt} (Line 12) before updating SS_{min}. Hence the values not in the found set will not be updated into SS_{temp}. The overall complexity of MIN is $O(\frac{nk(\tau+1)}{w\tau})$.

Algorithm 5 Bit-Parallel MIN in HBP

```
1: Initialize $SS_{temp}$: temp slot-wise minimum sub-segment
2: for each segment $S$ in column $C$ do
3:     for each sub-segment $SS_t$ in segment $S$ do
4:         Extract delimiter filter: $M_d := \leftarrow_{t-1} (S.F) \wedge 10^{\tau} \ldots 10^{\tau}$
5:         $SS_{temp} :=$ SUB-SLOTMIN$(SS_t, SS_{temp}, M_d)$
6: Reconstruct the values in $SS_{temp}$ to plain form
7: $min :=$ the minimum among the $\frac{w}{\tau+1}$ values
8: return $min$
9: procedure SUB-SLOTMIN($SS_x$, $SS_y$, $M_d$)
10:     Initialize $SS_{min}$
11:     $M_{lt} :=$ BIT-PARALLEL-LESSTHAN$(SS_x, SS_y)$
12:     $M_{lt} := M_{lt} \wedge M_d$
13:     $M := M_{lt} - \rightarrow_{\tau} (M_{lt})$
14:     for each word-group $G_i$ do
15:         $SS_{min}.W_i := (M \wedge SS_x.W_i) \vee (\neg M \wedge SS_y.W_i)$
16:     return $SS_{min}$
```

[MEDIAN] Without loss of generality, we again look for the lower median. Our idea is to progressively determine the value of the median bit-group by bit-group. The median $\mathcal{M}$ should be a k-bit value, divided into $\lceil \frac{k}{\tau} \rceil$ bit-groups. So, we denote the i-th bit-group of $\mathcal{M}$ as $\mathcal{M}.b_i$. For the example in Figure 4, $k = 6$ and the size of a bit-group $\tau = 3$. So, $\mathcal{M}.b_1$ and $\mathcal{M}.b_2$ refer to the three most and the three least significant bits of the median $\mathcal{M}$, respectively.

To determine $\mathcal{M}.b_1$ and $\mathcal{M}.b_2$, we build histograms. Specifically, we have designed a procedure BUILD--HISTOGRAM to build a cumulative histogram using every distinct bit-group as the bins. For the first word-group G_1 in Figure 4b, if we only focus on segment S_1, we have Step 1 as follows:

	Step 1			Step 2		
bin	member	freq	cumul' freq	member	freq	cumul' freq
000		0	0		0	0
001	$v_2^{(1)}$	1	1	$v_4^{(2)}$	1	1
010	$v_8^{(1)}$	1	2		0	1
011		0	2		0	1
100		0	2	$v_5^{(2)}$	1	**2**
101	$v_4^{(1)}, v_5^{(1)}, v_7^{(1)}$	3	**5**		0	2
110	$v_1^{(1)}$	1	6	$v_7^{(2)}$	1	3
111	$v_3^{(1)}, v_6^{(1)}$	2	8		0	3

As we are looking for the 4-th smallest value in the segment, from the histogram, we can assert that : (i) the median is among v_4, v_5, and v_7; (ii) the three most significant bits of the median, i.e., $\mathcal{M}.b_1$ must be $(101)_2$; (iii) the median is the 2-nd smallest among v_4, v_5, and v_7.

The next step is to examine the second word-group G_2 so as to determine $\mathcal{M}.b_2$. As such, we first exclude all values except v_4, v_5, and v_7 and then build the histogram (see Step 2 above). According to (iii) above, we are essentially looking for the 2-nd smallest value. Through the histogram, we can thus determine $\mathcal{M}.b_2 = 100_2$.

Algorithm 6 shows our implementation of MEDIAN of HBP. We initialize a candidate bit vector V to each segment as F to indicate median candidates. Then we determine the value of median $\mathcal{M}$ bit-group by bit-group (Lines 5–11). For each word-group G_i, we compute the histogram over the i-th bit-group of all candidates (Line 6). With the histogram we can identify the correct i-th bit-group value of $\mathcal{M}$ (Lines 7–8). The median's rank (r) is updated to reflect its ranking among the remaining candidates (Line 9). The next step is to update the candidate bit vector V of each segment (Lines 10–11), so that only tuples with the desired bit-group value (e.g., 101 in our example) are examined in the next round. To implement that, we use the BIT-PARALLEL-EQUAL procedure in [2].

The procedure BUILD-HISTOGRAM takes as input a word-group G_i and returns histogram (array) $HIST$ of all values' i-th bit-group. It iterates through all sub-segments. For each sub-segment, it extracts the delimiter filter M_d (refer to Step 1 of GET-VALUE-FILTER in Section III-B) belonging to that sub-segment (Line 19) and then examines the slots one by one (Lines 20–23). Similar to Algorithm 3, we may skip processing a whole segment by testing $S.V == 0$ (Line 16–17). Since the number of slots per word is given by $\frac{w}{\tau+1}$, the resulted complexity of the algorithm is $O(\frac{nk}{\tau})$.

We note that the histogram has 2^τ entries. So, in practice, when storing the data with bit-group, the size of the histogram is taken into account when selecting the value of τ so as to ensure the histogram can fit in the cache.

Algorithm 6 Bit-Parallel MEDIAN in HBP

```
1: Initialize $r = \lfloor \mathrm{COUNT}(*)/2 \rfloor$: the (lower) median's rank
2: Initialize $\mathcal{M} = 0$: the median
3: for each segment $S$ in column C do
4:     $S.V := S.F$
5: for each word-group $G_i$ do
6:     Histogram $HIST = \mathrm{BUILD\text{-}HISTOGRAM}(G_i)$
7:     $bin := \operatorname{argmin}_i \sum_{j=0}^{i} HIST[j] \geq r$
8:     $\mathcal{M}.b_i := bin$
9:     $r := r - \sum_{j=0}^{bin-1} HIST[j]$
10:    for each segment $S$ in column $C$ do
11:        $S.V := S.V \wedge \mathrm{BIT\text{-}PARALLEL\text{-}EQUAL}(G_i, bin)$
12: return $\mathcal{M}$
13: procedure BUILD-HISTOGRAM($G_i$)
14:    Initialize histogram $HIST[0 \ldots (2^\tau - 1)] := 0$
15:    for each segment $S$ in column $C$ do
16:        if $S.V == 0$ then
17:            break
18:        for each sub-segment $SS_t$ in segment $S$ do
19:            $M_d := (\leftarrow_{t-1} S.V) \wedge 10^\tau \ldots 10^\tau$
20:            for $i := 1 \ldots \frac{w}{\tau+1}$ do
21:                if the $i$-th delimiter in $M_d$ is 1 then
22:                    $v :=$ the $i$-th slot of $SS_t.W_i$
23:                    $HIST[v]++$
24:    return $HIST$
```

IV. Evaluation

We ran our experiments on a machine with a 3.40 GHz Intel i7-4770 quad-core CPU, and 16GB DDR3 memory. Each core has 32KB L1i cache, 32KB L1d cache and 256KB L2 unified cache. All cores share an 8MB L3 cache. The CPU supports AVX2 instruction set that operates on 256-bit SIMD registers. We compare our bit-parallel (BP) aggregation algorithms with the non-bit-parallel (NBP) aggregation approach mentioned in the beginning of Section III. Both approaches take as input a filter bit vector F and return an aggregate value. All algorithms were implemented in C++ and compiled using g++ 4.9 with optimization flag -O3.

A. Micro-Benchmark Evaluation

We evaluate the performance by varying (1) the selectivity, (2) the value width, and (3) the data size in this evaluation. The default data size, value width, CPU word width and selectivity are one billion tuples, $k = 25$, $w = 64$ and 0.1, respectively. The bit-group size τ was empirically determined, according to [2][4]. We report only the performance of SUM, MIN/MAX, and MEDIAN as the implementation of COUNT is straightforward and the performance of AVG is the sum of SUM plus COUNT. Following [2], we use a benchmark query like Q1 (Section III) to conduct the experiments. In this micro-benchmark evaluation, we ran experiments using a single thread. We have profiled our algorithms using Intel's VTune profiler (https://software.intel.com/en-us/intel-vtune-amplifier-xe). We observed an average CPI (cycles per instruction) of 0.5 and less than 20% CPU cycles are stalled due to L3 cache miss on average. Therefore, we conclude our algorithms are CPU bound and optimizing the aggregation phase is worth doing.

Varying Selectivity Recall that the non-bit-parallel (NBP) aggregation methods burn instructions by reconstructing data values that pass the filter whereas our bit-parallel (BP) aggregation algorithms avoid that. In this experiment, our goal is to study how would the improvement of our BP aggregation methods over the NBP aggregation methods get affected when fewer/more tuples pass the filter. Figure 5 shows the speed-up of the aggregation phase using our BP aggregation methods versus using the NBP aggregation methods in terms of number of cycles-per-tuple. The number of processor cycles is measured using the RDTSC instruction. We remark that the cycles per tuple reported in our experiments is equivalent to the wall clock time, that also implies the overall running time

[4]In [2], the optimal τ was 4 for VBP. We adopted that in our experiment because we also found the same optimal value in our empirical experiments. The authors of [2] found that using bit-group was not fruitful under HBP and they thus set $\tau = k$ (i.e., use no bit-group). However, we found that bit-group was actually helpful when the programs were carefully optimized. Furthermore, we found that the optimal τ for different k and w values on HBP can actually be analytically determined. Hence, in this paper, we adopted the optimal τ values found by our analytical technique. Note that such optimal value improves the performance of both the baseline methods and our methods, so it is a fair comparison. Readers interest in that analytical equation and the program optimizations are referred to [14].

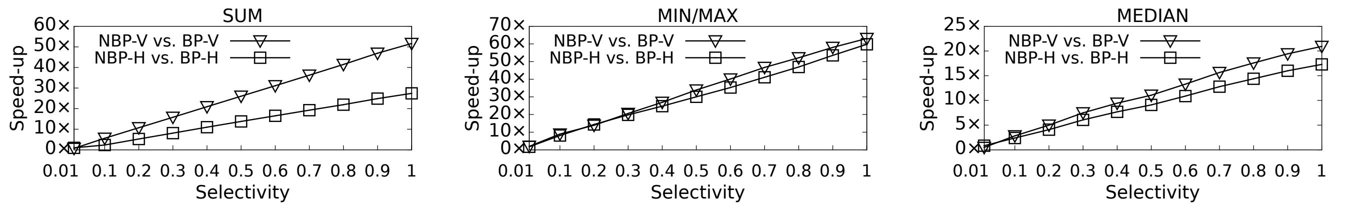

Fig. 5. Varying selectivity vs. Cost of aggregation

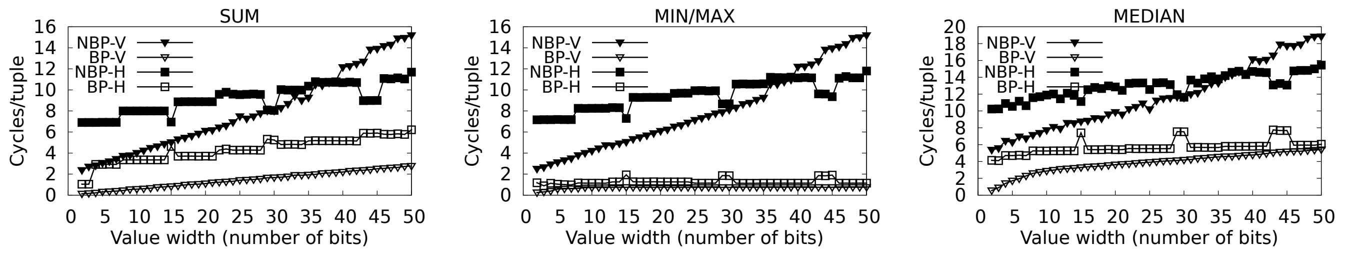

Fig. 6. Varying value width vs. Cost of aggregation

of the algorithms, but not only the CPU costs. We varied the selectivity from 0.01 (selective) to 1. It shows that our BP approach offers significant speed-up over the NBP methods and the speed-up increases with the selectivity. For example, at selectivity 0.1, the NBP methods are shown to use $4\times$, $8.5\times$, and $2.6\times$ cycles more than our BP methods when carrying out `SUM`, `MIN/MAX`, and `MEDIAN`, respectively. The speed-up over NBP methods for `MIN/MAX` is higher than for `SUM` and `MEDIAN` because `MIN/MAX` can enjoy early-stopping when values are pruned. In contrast, `SUM` intrinsically needs all values and thus no early-stopping. Similarly, `MEDIAN` is well-known as a holistic measure where the computation requires the entire data set. Therefore, it also cannot enjoy early stopping as `MIN` and `MAX` as do.

Varying Value Width Bit-parallel algorithms have a property that their running time basically decreases when more values can be packed into a word (i.e., a higher degree of parallelism). In this experiment, our goal is to study how the value width k impacts the performance of our methods. Figure 6 shows the cycles-per-tuple of NBP aggregation methods and our BP aggregation methods under the default data size (1 billion tuple) and default selectivity (0.1) during the aggregation phase. The value width k was varied from 2 bits to 50 bits. We can see that our BP aggregation methods outperform the NBP methods under all value widths. Generally, all methods consume more cycles when the value width increases because of the decrease in parallelism. HBP-based algorithms, be it a BP method or a NBP method, have the property of one iteration per τ-bits. VBP-based algorithms, in contrast, have the property of one iteration per bit. That explains why the increase under HBP is milder than the increase under VBP in Figure 6. Finally, in HBP, we can still achieve certain parallelism for $k \geq w/2$ because of bit-groups. Specifically, when $k \geq w/2$, without bit-groups, HBP-based solutions will downgrade to the naive format (one data item per word) and gain no benefit from bit-parallelism and early stopping. However, with bit-groups, long values are split up such that parallelism still applies.

Varying Data Size In this experiment we aim to see how our BP algorithms scale according to the data size. We varied the data size from one to four billion tuples. Basically our complexity analysis for each BP algorithm has shown that our methods scale linearly with the data size. Figure 7 confirms that. Furthermore, we see from the figure that our BP algorithms can provide absolute time improvement over the NBP algorithms up to 10 seconds (for `MIN/MAX`), which is significant to main-memory query processing.

B. Multi-threading and SIMD Acceleration

We have also implemented multi-threading and SIMD acceleration for our algorithms. For multi-threading, we used four threads with each pinned to a physical core of our CPU and let each thread process a partition of data. Multi-threading is associated with synchronization overheads though. For example, in VBP-MEDIAN, all threads must synchronize their access to the global counter c after each iteration (see Line 8 in Algorithm 3). Multi-threading is implemented with OpenMP.

For SIMD acceleration, we exploit the AVX2 instruction set in our CPU. The SIMD code is written using intrinsics. For VBP, it simply views 256-bit SIMD registers as 256-bit CPU words, because its algorithms use only bitwise instructions such as AND, OR, XOR. Thus, a VBP segment now contains 256 values. For HBP, it relies on shifts and additions. SIMD shifts and additions can operate on four 64-bit banks in parallel, but not the 256-bit word as a whole. Thus, we run four instances of 64-bit algorithms in the 256-bit SIMD registers. As a result, we process four segments in parallel and each segment is self-contained in its 64-bit boundary. We leveraged the vectorized instructions whenever possible. For example, when appropriate, we used the `_mm256_cmpgt_epi8` instruction to compare 32 pairs of 8-bit values at the same time.

Figure 8 shows the speed-up of our BP algorithms by enabling multi-threading and/or SIMD acceleration. The shaded bars show the speed-up of our BP algorithms by enabling both

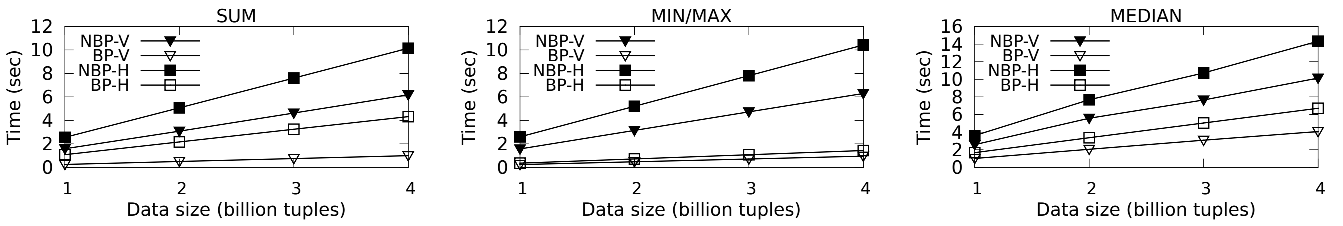

Fig. 7. Varying data size vs. Cost of aggregation

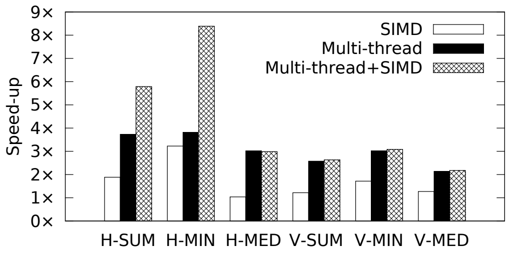

Fig. 8. Speed-up over single-threaded bit-parallel implementation by multi-threading and SIMD

multi-threading and SIMD acceleration. We see a speed-up from $2.2\times$ to $8.4\times$ overall.

The black bars in Figure 8 show the speed-up of our BP algorithms by enabling multi-threading. We see that enabling multi-threading generally makes our BP algorithms to run about $2.1\times$ to $3.8\times$ faster. The speed-up cannot reach $4\times$ because there are synchronization overheads in using multi-threading.

The white bars in Figure 8 show the speed-up of our BP algorithms by enabling SIMD acceleration. We see that SIMD generally makes our BP algorithms run up to $3.2\times$ faster. It is generally better than the marginal benefit, 10% speed-up, of using SIMD reported in [2]. Nevertheless, the speed-up cannot reach $4\times$. One reason is that SIMD instructions generally consume more cycles per instruction (around $1.5\times$) than their non-SIMD counterparts [2]. Another reason is that not every instruction in 64-bit generic ALU has its 256-bit counterpart in SIMD. For example, at the moment AVX2 has not yet provided a 256-bit `POPCNT` instruction. Therefore, a 256-bit `POPCNT` is essentially translated into to four serial 64-bit `POPCNT` calls. Consequently, the extra parallelism of SIMD cannot be fully unleashed, especially when our BP algorithms rely on `POPCNT`. This also explains why the speed-up of using SIMD in HBP is greater than in VBP because our BP algorithms use `POPCNT` more often in VBP than in HBP. Nevertheless, we believe this `POPCNT` issue will be resolved after the future SIMD instruction set includes the 256-bit `POPCNT` instruction.

C. TPC-H Evaluation

Lastly, we evaluate the overall effectiveness of our algorithms using TPC-H benchmark at scale factor 10GB. 10 out of 22 TPC-H queries have selectivity higher than 0.01, but we discard Q4 because it involves only straightforward COUNT aggregation. We follow [11], [12] so that complex queries can be transformed into simple filter scans and simple aggregation. The filter scans are implemented using bit-parallel algorithms in [2].

Table II shows the experimental results. We see that by using our bit-parallel algorithms, the cycles (time) spent on aggregation can be reduced by 28.1% (HBP) and 55.0% (VBP) on average (see the **bold** lines in the table). So now, with our bit-parallel algorithms, the aggregation operations are no longer much slower than the bit-parallel scanning operators. Overall, with our bit-parallel algorithms, the total query execution cost can be reduced by 20.4% (HBP) and 44.4% (VBP) on average. The significant time reduction for those long queries (e.g., from 86 cycles down to 2.4 cycles for Q1) especially matches our goal of reducing the running times of long queries.

V. Related Work

Recently, there is an increasing number of studies attempt to take advantage of modern hardware to speed up query processing. One line of work is to offload some operators to the graphics processing unit (GPU) when processing queries [15], [8], [9], [16]. A GPU contains many SIMD multi-processors, providing intrinsic massive parallelism. It also possesses a large device memory and has high on-chip memory bandwidth. The development of GPGPU (General Purpose GPU) languages in recent years such as NVIDIA CUDA has made general programming using GPU much simpler. However, the potential of utilizing GPU may be hurdled by the slow communication between GPU and CPU via PCI bus. Hence, it requires special attention when designing algorithms on GPU. Typical relational operations like selection, sort and join on GPU have been studied in [8], [9]. It is reported that computational expensive operators like joins can benefit significantly from the parallelism of GPU.

Another line of work is to leverage the SIMD instruction set in modern processors so as to parallelize database operations [17], [18], [19], [20], [21]. An SIMD register is typically 128/256 bits long. The processor can execute a single instruction stream on multiple n-bit operands, where $n = 8, 16, 32, 64$, etc.

There is another line of work that exploits multi-core architecture in modern CPUs to parallelize aggregation. For example, multi-threaded aggregation on modern multi-core processor is studied in [22], [23], [24]. Their challenge is to tackle resource contention among the increasing number of threads.

To some extent, the studies mentioned above can be classified as "inter-word" parallelism, as they typically process one data item stored in one word. In contrast, Blink [6], [25], [1], BitWeaving (the collective term for performing scans on HBP and VBP storage layout) [2], and this paper drill down to the level of "intra-word" parallelism. Both Blink

TABLE II. EXPERIMENTS ON TPC-H QUERIES (10GB DATA; MULTI-THREADED; SIMD-ENABLED).

Query	Q1		Q6		Q7		Q9		Q10		Q11		Q14		Q15		Q20		
Selectivity	0.986		0.019		0.301		0.053		0.019		0.041		0.012		0.037		0.150		
HBP	Reported cost is cycles per tuple, equivalent to running time.																		
BP filter scan	0.4	0.4	1.5	1.5	0.8	0.8	0.4	0.4	1.0	1.0	0.1	0.1	0.8	0.8	0.8	0.8	0.9	0.9	
Aggregation (NBP | BP)	21.7	2.3	0.6	0.6	2.1	0.8	0.9	0.7	0.6	0.6	1.1	1.0	0.5	0.5	0.7	0.7	0.5	0.2	Avg
Aggregation improvement*	**89.4%**		**0.7%**		**62.5%**		**22.9%**		**0.3%**		**12.3%**		**3.0%**		**3.4%**		**58.4%**		**28.1%**
Total cost (NBP | BP)	22.1	2.7	2.1	2.1	2.9	1.6	1.3	1.1	1.7	1.7	1.2	1.1	1.3	1.3	1.4	1.4	1.3	1.1	Avg
Overall improvement*	87.7%		0.2%		45.8%		15.1%		0.1%		11.4%		1.2%		1.6%		20.4%		20.4 %
VBP	Reported cost is cycles per tuple, equivalent to running time.																		
BP filter scan	0.4	0.4	1.1	1.1	0.6	0.6	0.4	0.4	0.8	0.8	0.0	0.0	0.6	0.6	0.6	0.6	0.7	0.7	
Aggregation (NBP | BP)	85.6	2.0	0.7	0.6	8.0	0.8	2.1	0.6	0.7	0.6	1.6	0.8	0.6	0.6	1.2	0.6	1.1	0.2	Avg
Aggregation improvement*	**97.7%**		**20.2%**		**90.1%**		**72.0%**		**20.8%**		**50.3%**		**6.4%**		**51.3%**		**86.3%**		**55.0%**
Total cost (NBP | BP)	86.0	2.4	1.8	1.6	8.7	1.4	2.4	0.9	1.5	1.4	1.7	0.8	1.2	1.2	1.8	1.2	1.9	0.9	Avg
Overall improvement*	97.3%		8.2%		83.5%		61.5%		10.1%		49.2%		3.2%		33.7%		53.1%		44.4%

*Improvement = (NBP time - BP time)/NBP time $\times 100\%$

and BitWeaving are general solutions to parallelizing scan operations on commodity CPUs. IBM Blink system is a row-store whereas BitWeaving assumes a column-store. Indeed, the VBP layout in BitWeaving was inspired by bit-slice [10], [26] whereas the HBP layout in Bitweaving was pioneered by Lamport [27]. Once again, we remark that Blink, Bitweaving, and our techniques here in this paper can integrate with the three lines of hardware-specific techniques above to enjoy extra speed-up, as exemplified by our experiments above (Section IV-B).

VI. CONCLUSION

Aggregation is an important operation in main memory analytical processing engine. Recently, there are studies to speed up filter scan in main memory databases by fully utilizing the intra-cycle parallelism in modern CPUs [1], [2]. In this paper, we showed that, with efficient intra-cycle parallel scan, aggregations `SUM, MIN, MAX, AVG, MEDIAN` and `COUNT` would become the bottleneck if they do not leverage the CPU's intra-cycle parallelism accordingly. Therefore, in this paper, we contribute a suite of efficient aggregation algorithms that also exploit intra-cycle parallelism. Our algorithms use no specialized but standard instruction sets found in all modern CPU architecture. Furthermore, our algorithms can integrate with SIMD instruction sets and multi-threading to enjoy further speed-up. Experiment results show that our algorithms yield significant speed up to all aggregate operations.

REFERENCES

[1] R. Johnson, V. Raman, R. Sidle, and G. Swart, "Row-wise parallel predicate evaluation," *PVLDB*, 2008.

[2] Y. Li and J. M. Patel, "Bitweaving: Fast scans for main memory data processing," in *SIGMOD*, 2013.

[3] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. Kersten, "Monetdb: Two decades of research in column-oriented database architectures," *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 2012.

[4] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The SAP HANA Database–An Architecture Overview." *IEEE Data Eng. Bull.*, 2012.

[5] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *SIGMOD*, 2006.

[6] R. Barber, P. Bendel, M. Czech, O. Draese, F. Ho, N. Hrle, S. Idreos, M.-S. Kim, O. Koeth, J.-G. Lee *et al.*, "Business analytics in (a) blink." *IEEE Data Eng. Bull.*, 2012.

[7] W. Fang, B. He, and Q. Luo, "Database compression on graphics processors," *PVLDB*, 2010.

[8] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander, "Relational query coprocessing on graphics processors," *TODS*, 2009.

[9] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational joins on graphics processors," in *SIGMOD*, 2008.

[10] P. O'Neil and D. Quass, "Improved query performance with variant indexes," in *ACM SIGMOD Record*, 1997.

[11] Y. Li and J. M. Patel, "Widetable: An accelerator for analytical data processing," *PVLDB*, 2014.

[12] A. Lamb, M. Fuller, R. Varadarajan, N. Tran, B. Vandiver, L. Doshi, and C. Bear, "The vertica analytic database: C-store 7 years later," *PVLDB*, 2012.

[13] M. V. Wilkes, D. J. Wheeler, and S. Gill, *The Preparation of Programs for an Electronic Digital Computer (Charles Babbage Institute Reprint)*, 1984.

[14] Z. Feng and E. Lo, "Accelerating aggregation using intra-cycle parallelism (technical report)," 2014, http://www4.comp.polyu.edu.hk/ cszqfeng/pub/bpagg.pdf.

[15] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast computation of database operations using graphics processors," in *SITMOD*, 2004.

[16] J. He, M. Lu, and B. He, "Revisiting co-processing for hash joins on the coupled cpu-gpu architecture," *PVLDB*, 2013.

[17] J. Zhou and K. A. Ross, "Implementing database operations using SIMD instructions," in *SIGMOD*, 2002.

[18] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner, "SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units," *PVLDB*, 2009.

[19] T. Willhalm, I. Oukid, I. Müller, and F. Faerber, "Vectorizing database column scans with complex predicates," in *ADMS Workshop*, 2013.

[20] J. Chhugani, A. D. Nguyen, V. W. Lee, W. Macy, M. Hagog, Y.-K. Chen, A. Baransi, S. Kumar, and P. Dubey, "Efficient implementation of sorting on multi-core SIMD CPU architecture," *PVLDB*, 2008.

[21] C. Balkesen, G. Alonso, and M. Ozsu, "Multi-core, main-memory joins: Sort vs. hash revisited," *PVLDB*, 2013.

[22] J. Cieslewicz and K. A. Ross, "Adaptive aggregation on chip multiprocessors," in *PVLDB*, 2007.

[23] J. Cieslewicz, K. A. Ross, K. Satsumi, and Y. Ye, "Automatic contention detection and amelioration for data-intensive operations," in *SIGMOD*, 2010.

[24] Y. Ye, K. A. Ross, and N. Vesdapunt, "Scalable aggregation on multicore processors," in *DeMoN*, 2011.

[25] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle, "Constant-time query processing," in *ICDE*, 2008.

[26] D. Rinfret, P. O'Neil, and E. O'Neil, "Bit-sliced index arithmetic," in *ACM SIGMOD Record*, 2001.

[27] L. Lamport, "Multiple byte processing with full-word instructions," *Commun. ACM*, 1975.