Rethinking SIMD Vectorization for In-Memory Databases

Orestis Polychroniou*
Columbia University
orestis@cs.columbia.edu

Arun Raghavan
Oracle Labs
arun.raghavan@oracle.com

Kenneth A. Ross†
Columbia University
kar@cs.columbia.edu

ABSTRACT

Analytical databases are continuously adapting to the underlying hardware in order to saturate all sources of parallelism. At the same time, hardware evolves in multiple directions to explore different trade-offs. The MIC architecture, one such example, strays from the mainstream CPU design by packing a larger number of simpler cores per chip, relying on SIMD instructions to fill the performance gap. Databases have been attempting to utilize the SIMD capabilities of CPUs. However, mainstream CPUs have only recently adopted wider SIMD registers and more advanced instructions, since they do not rely primarily on SIMD for efficiency. In this paper, we present novel vectorized designs and implementations of database operators, based on advanced SIMD operations, such as gathers and scatters. We study selections, hash tables, and partitioning; and combine them to build sorting and joins. Our evaluation on the MIC-based Xeon Phi co-processor as well as the latest mainstream CPUs shows that our vectorization designs are up to an order of magnitude faster than the state-of-the-art scalar and vector approaches. Also, we highlight the impact of efficient vectorization on the algorithmic design of in-memory database operators, as well as the architectural design and power efficiency of hardware, by making simple cores comparably fast to complex cores. This work is applicable to CPUs and co-processors with advanced SIMD capabilities, using either many simple cores or fewer complex cores.

1. INTRODUCTION

Real time analytics are the steering wheels of big data driven business intelligence. Database customer needs have extended beyond OLTP with high ACID transaction throughput, to interactive OLAP query execution across the entire database. As a consequence, vendors offer fast OLAP solutions, either by rebuilding a new DBMS for OLAP [28, 37], or by improving within the existing OLTP-focused DBMS.

The advent of large main-memory capacity is one of the reasons that blink-of-an-eye analytical query execution has become possible. Query optimization used to measure blocks fetched from disk as the primary unit of query cost. Today, the entire database can often remain main-memory resident and the need for efficient in-memory algorithms is apparent.

The prevalent shift in database design for the new era are column stores [19, 28, 37]. They allow for higher data compression in order to reduce the data footprint, minimize the number of columns accessed per tuple, and use column oriented execution coupled with late materialization [9] to eliminate unnecessary accesses to RAM resident columns.

Hardware provides performance through three sources of parallelism: *thread parallelism*, *instruction level parallelism*, and *data parallelism*. Analytical databases have evolved to take advantage of all sources of parallelism. Thread parallelism is achieved, for individual operators, by splitting the input equally among threads [3, 4, 5, 8, 14, 31, 40], and in the case of queries that combine multiple operators, by using the pipeline breaking points of the query plan to split the materialized data in chunks that are distributed to threads dynamically [18, 28]. Instruction level parallelism is achieved by applying the same operation to a block of tuples [6] and by compiling into tight machine code [16, 22]. Data parallelism is achieved by implementing each operator to use SIMD instructions effectively [7, 15, 26, 30, 39, 41].

The different sources of parallelism were developed as a means to deliver more performance within the same power budget available per chip. Mainstream CPUs have evolved on all sources of parallelism, featuring massively superscalar pipelines, out-of-order execution of tens of instructions, and advanced SIMD capabilities, all replicated on multiple cores per CPU chip. For example, the latest Intel Haswell architecture issues up to 8 micro-operations per cycle with 192 reorder buffer entries for out-of-order execution, 256-bit SIMD instructions, two levels of private caches per core and a large shared cache, and scales up to 18 cores per chip.

Concurrently with the evolution of mainstream CPUs, a new approach on processor design has surfaced. The design, named the many-integrated-cores (MIC) architecture, uses cores with a smaller area (transistors) and power footprint by removing the massively superscalar pipeline, out-of-order execution, and the large L3 cache. Each core is based on a Pentium 1 processor with a simple in-order pipeline, but is augmented with large SIMD registers, advanced SIMD instructions, and simultaneous multithreading to hide load and instruction latency. Since each core has a smaller area and power footprint, more cores can be packed in the chip.

*Work partly done when first author was at the Oracle Labs.
†Supported by NSF grant IIS-1422488 and an Oracle gift.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

http://dx.doi.org/10.1145/2723372.2747645.

The MIC design was originally intended as a GPU [33], but now targets high performance computing applications. Using a high FLOPS machine to execute compute-intensive algorithms with superlinear complexity is self-evident. Executing analytical queries in memory, however, consists of data-intensive linear algorithms that mostly "move" rather than "process" data. Previous work to add SIMD in databases has optimized sequential access operators such as index [41] or linear scans [39], built multi-way trees with nodes that match the SIMD register layout [15, 26], and optimized specific operators, such as sorting [7, 11, 26], by using *ad-hoc* vectorization techniques, useful only for a specific problem.

In this paper, we present good design principles for SIMD vectorization of main-memory database operators, without modifying the logic or the data layout of the baseline scalar algorithm. The baseline algorithm is defined here as the most straightforward scalar implementation. Formally, assume an algorithm that solves a problem with optimal complexity, its simplest scalar implementation, and a vectorized implementation. We say that the algorithm can be *fully vectorized*, if the vector implementation executes $O(f(n)/W)$ vector instructions instead of $O(f(n))$ scalar instructions where W is the vector length, excluding random memory accesses that are by definition not data-parallel. We define fundamental vector operations that are frequently reused in the vectorizations and are implemented using advanced SIMD instructions, such as non-contiguous loads (*gathers*) and stores (*scatters*). The fundamental operations that are not directly supported by specific instructions can be implemented using simpler instructions at a performance penalty.

We implement vectorized operators in the context of main-memory databases: selection scans, hash tables, and partitioning, which are combined to build more advanced operators: sorting and joins. These operators cover a large portion of the time needed to execute analytical queries in main memory. For selection scans, we show branchless tuple selection and in-cache buffering. For hash tables, we study both building and probing across using multiple hashing schemes. For partitioning, we describe histogram generation, including all partitioning function types: radix, hash, and range. We also describe data shuffling, including inputs larger than the cache. All of the above are combined to build radixsort and multiple hash join variants that highlight the impact of vectorization on determining the best algorithmic design.

We compare our vectorized implementations of in-memory database operators against the respective state-of-the-art scalar and vector techniques, by evaluating on the Intel Xeon Phi co-processor and on the latest mainstream CPUs. Xeon Phi is currently the only available hardware based on the MIC architecture and is also the only generally available hardware that supports *gathers* and *scatters*, while the latest mainstream CPUs (Intel Haswell) support gathers.

We use the sorting and join operator to compare a Xeon Phi 7120P co-processor (61 P54C cores at 1.238 GHz, 300 Watts TDP) against four high-end Sandy Bridge CPUs (4×8 Sandy Bridge cores at 2.2 GHz, 4 × 130 Watts TDP), and found that they have similar performance, but on a different power budget, since Xeon Phi spends almost half the energy.

The next generation of Xeon Phi will also be available as a standalone CPU,[1] even if the current generation is only available as a co-processor, and will support more advanced SIMD instructions (AVX 3), also supported by the next generation of mainstream CPUs.[2] Our work does not focus on evaluating Xeon Phi as a co-processing accelerator, such as GPUs, that would also be bound by the PCI-e bandwidth, but as an alternative CPU design that is suitable and more power efficient for executing analytical database workloads.

We summarize our contributions:

- We introduce design principles for efficient vectorization of in-memory database operators and define fundamental vector operations that are frequently reused.
- We design and implement vectorized selection scans, hash tables, and partitioning, that are combined to design and build sorting and multiple join variants.
- We compare our implementations against state-of-the-art scalar and vectorized techniques. We achieve up to an order of magnitude speedups by evaluating on Xeon Phi as well as on the latest mainstream CPUs.
- We show the impact of vectorization on the algorithmic design of in-memory operators, as well as the architectural design and power efficiency of hardware, by making simple cores comparably fast to complex cores.

The rest of the paper is organized as follows. Section 2 presents related work. Sections 4, 5, 6, and 7 discuss the vectorization of selection scans, hash tables, Bloom filters, and partitioning. Sections 8 and 9 discuss algorithmic designs for sorting and hash join. We present our experimental evaluation in Section 10, we discuss how SIMD vectorization relates to GPUs in Section 11, and conclude in Section 12. Implementation details are provided in the Appendix.

2. RELATED WORK

Previous work that added SIMD instructions in database operators is briefly summarized. Zhou et al. used SIMD for linear scans, index scans, and nested loop joins [41]. Ross proposed probing multiple keys per bucket for cuckoo hashing [30]. Willhalm et al. optimized decompression of bit packed data [39]. Inoue et al. proposed data-parallel combsort and merging [11]. Chhugani et al. optimized bitonic merging for mergesort [7]. Kim et al. designed multi-way trees tailored to the SIMD layout [15]. Polychroniou et al. discussed trade-offs for updating heavy hitter aggregates [25], fast range partitioning via a range function index [26], and Bloom filter probing using gathers [27]. Schlegel et al. described scalable frequent itemset mining on Xeon Phi [32].

Database operators have been extensively optimized for modern processors. Manegold et al. introduced hash joins with cache-conscious partitioning [19], which Kim et al. extended to multi-core CPUs and compared against SIMD sort-merge joins [14]. Cieslewicz et al. and Ye et al. studied contention-free aggregation [8, 40]. Blanas et al. and Balkesen et al. evaluated hardware-tuned hash joins [3, 5]. Albutiu et al. introduced NUMA-aware joins and Balkesen et al. evaluated join variants on multiple CPUs [1, 4]. Satish et al. and Wassenberg et al. introduced buffered partitioning for radixsort [31, 38]. Polychroniou et al. introduced in-place and range partitioning for sorting variants [26]. Jha et al. optimized hash joins on Xeon Phi, but used SIMD partially only to compute hash functions and load hash buckets [12].

[1] newsroom.intel.com/community/intel_newsroom/blog/2014/06/23/intel-re-architects-the-fundamental-building-block-for-high-performance-computing

[2] software.intel.com/blogs/2013/avx-512-instructions

Compilers partially transform scalar to SIMD code based on loop unrolling [17] and control flow predication [34]. Our designs are far more complicated and not strictly equivalent. Database operators can be compiled directly [22], thus manual vectorization is also desirable to maximize performance.

GPUs have also been used in databases for generic queries [2] and to accelerate operators, such as indexes [15], sorting [31], joins [13, 24], and selections [36]. SIMT GPU threads are organized in *warps* that execute the same scalar instruction per cycle by predicating all control flow. In SIMD code, control flow is converted to data flow in software. We discuss the similarities between SIMD and SIMT code and to what extent our work is applicable to SIMT GPUs in Section 11.

3. FUNDAMENTAL OPERATIONS

In this section we define the fundamental vector operations that we will need to implement vectorized database operators. The first two operations, termed *selective load* and *selective store*, are spatially contiguous memory accesses that load or store values using a *subset* of vector lanes. The last two operations are spatially non-contiguous memory loads and stores, termed *gathers* and *scatters* respectively.

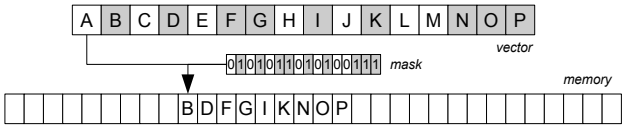

Figure 1: Selective store operation

Selective stores write a specific subset of the vector lanes to a memory location contiguously. The subset of vector lanes to be written is decided using a vector or scalar register as the mask, which must not be limited to a constant.

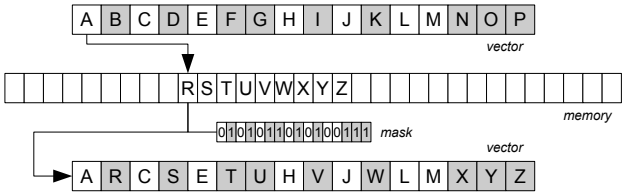

Figure 2: Selective load operation

Selective loads are the symmetric operation that involves loading from a memory location contiguously to a subset of vector lanes based on a mask. The lanes that are inactive in the mask retain their previous values in the vector.

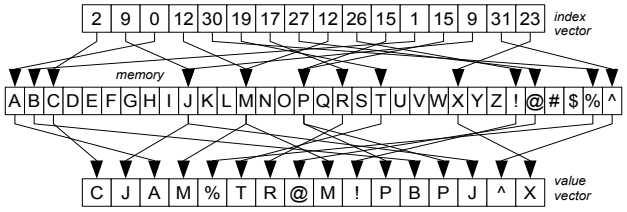

Figure 3: Gather operation

Gather operations load values from non-contiguous locations. The inputs are a vector of indexes and an array pointer. The output is a vector with the values of the respective array cells. By adding a mask as an input operand, we define the selective gather that operates on a subset of lanes. The inactive mask lanes retain their previous contents.

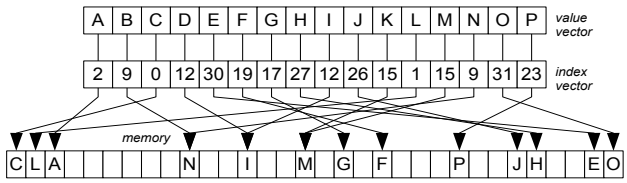

Figure 4: Scatter operation

Scatter operations execute stores to multiple locations. The input is a vector of indexes, an array pointer, and a vector of values. If multiple vector lanes point to the same location, we assume that the rightmost value will be written. By adding a mask as an input we can store lanes selectively.

Gathers and scatters are not really executed in parallel because the (L1) cache allows one or two distinct accesses per cycle. Executing W cache accesses per cycle is an impractical hardware design. Thus, random memory accesses have to be excluded from the $O(f(n)/W)$ vectorization rule.

Gathers are supported on the latest mainstream CPUs (Haswell) but scatters are not. Older mainstream CPUs (e.g., Sandy Bridge) support neither. Emulating gathers is possible at a performance penalty, which is small if done carefully. We discuss more hardware details in Appendix B.

Selective loads and stores are also not supported on the latest mainstream CPUs, but can be emulated using vector permutations. The lane selection mask is extracted as a bitmask and is used as an array index to load a permutation mask from a pre-generated table. The data vector is then permuted in a way that splits the active lanes of the mask to the one side of the register and the inactive lanes to the other side. In case of a selective store we can store the vector (unaligned) and in case of a selective load, we load a new vector (unaligned) and blend the two vectors to replace the inactive lanes. This technique was first used in vectorized Bloom filters [27] on CPUs, without defining the operations. We describe the Xeon Phi instructions in Appendix C.

4. SELECTION SCANS

Selection scans have re-emerged for main-memory query execution and are replacing traditional unclustered indexes in modern OLAP DBMSs [28]. Advanced optimizations include lightweight bit compression [39] to reduce the RAM bandwidth, generation of statistics to skip data regions [28], and scanning of bitmaps-zonemaps to skip cache lines [35].

Linear selective scan performance has been associated with branch mispredictions, if the operator is implemented as shown in Algorithm 1. Previous work has shown that converting control flow to data flow can affect performance, making different approaches optimal per selectivity rate [29]. Branches can be eliminated as shown in Algorithm 2 to avoid misprediction penalties, at the expense of accessing all payload columns and eagerly evaluating all selective predicates.

Algorithm 1 Selection Scan (Scalar - Branching)

```
j ← 0                                                  ▷ output index
for i ← 0 to |T_keys_in| − 1 do
    k ← T_keys_in[i]                                   ▷ access key columns
    if (k ≥ k_lower) && (k ≤ k_upper) then             ▷ short circuit and
        T_payloads_out[j] ← T_payloads_in[i]           ▷ copy all columns
        T_keys_out[j] ← k
        j ← j + 1
    end if
end for
```

Algorithm 2 Selection Scan (Scalar - Branchless)

```
j ← 0                                                    ▷ output index
for i ← 0 to |T_keys_in| − 1 do
    k ← T_keys_in[i]                                     ▷ copy all columns
    T_payloads_out[j] ← T_payloads_in[i]
    T_keys_out[j] ← k
    m ←(k ≥ k_lower ? 1 : 0) & (k ≤ k_upper ? 1 : 0)
    j ← j + m        ▷ if-then-else expressions use conditional ...
end for          ▷ ... flags to update the index without branching
```

Vectorized selection scans use selective stores to store the lanes that satisfy the selection predicates. We use SIMD instructions to evaluate the predicates resulting in a bitmask of the qualifying lanes. Partially vectorized selection extracts one bit at a time from the bitmask and accesses the corresponding tuple. Instead, we use the bitmask to selectively store the qualifying tuples to the output vector at once.

When the selection has a very low selectivity, it is desirable to avoid accessing the payload columns due to the performance drop caused by the memory bandwidth. Furthermore, when the branch is speculatively executed, we issue needless loads to payloads. To avoid reducing the bandwidth, we use a small cache resident buffer that stores indexes of qualifiers rather than the actual values. When the buffer is full, we reload the indexes from the buffer, gather the actual values from the columns, and flush them to the output. This variant is shown in Algorithm 3. Appendix A describes the notation used in the algorithmic descriptions.

When we materialize data on RAM without intent to reuse them soon, we use streaming stores. Mainstream CPUs provide non-temporal stores that bypass the higher cache levels and increase the RAM bandwidth for storing data. Xeon Phi does not support scalar streaming stores, but provides an instruction to overwrite a cache line with data from a vector without first loading it. This technique requires the vector length to be equal to the cache line and eliminates the need for write-combining buffers used in mainstream CPUs. All operators that write the output to memory sequentially, use buffering, which we omit in the algorithmic descriptions.

Algorithm 3 Selection Scan (Vector)

```
i, j, l ← 0                                  ▷ input, output, and buffer indexes
r⃗ ← {0, 1, 2, 3, ..., W − 1}                  ▷ input indexes in vector
for i ← 0 to |T_keys_in| − 1 step W do          ▷ # of vector lanes
    k⃗ ← T_keys_in[i]                          ▷ load vectors of key columns
    m ← (k⃗ ≥ k_lower) & (k⃗ ≤ k_upper)          ▷ predicates to mask
    if m ≠ false then                                ▷ optional branch
        B[l] ←_m r⃗                             ▷ selectively store indexes
        l ← l + |m|                                ▷ update buffer index
        if l > |B| − W then                              ▷ flush buffer
            for b ← 0 to |B| − W step W do
                p⃗ ← B[b]                           ▷ load input indexes
                k⃗ ← T_keys_in[p⃗]                  ▷ dereference values
                v⃗ ← T_payloads_in[p⃗]
                T_keys_out[b + j] ← k⃗      ▷ flush to output with ...
                T_payloads_out[b + j] ← v⃗  ▷ ... streaming stores
            end for
            p⃗ ← B[|B| − W]                        ▷ move overflow ...
            B[0] ← p⃗                          ▷ ... indexes to start
            j ← j + |B| − W                      ▷ update output index
            l ← l − |B| + W                      ▷ update buffer index
        end if
    end if
    r⃗ ← r⃗ + W                                  ▷ update index vector
end for                          ▷ flush last items after the loop
```

5. HASH TABLES

Hash tables are used in database systems to execute joins and aggregations since they allow constant time key lookups. In hash join, one relation is used to build the hash table and the other relation probes the hash table to find matches. In group-by aggregation they are used either to map tuples to unique group ids or to insert and update partial aggregates.

Using SIMD instructions in hash tables has been proposed as a way to build bucketized hash tables. Rather than comparing against a single key, we place multiple keys per bucket and compare them to the probing key using SIMD vector comparisons. We term the approach of comparing a single input (probing) key with multiple hash table keys, *horizontal vectorization*. Some hash table variants such as bucketized cuckoo hashing [30] can support much higher load factors. Loading a single 32-bit word is as fast as loading an entire vector, thus, the cost of bucketized probing diminishes to extracting the correct payload, which requires $\log W$ steps.

Horizontal vectorization, if we expect to search fewer than W buckets on average per probing key, is wasteful. For example, a 50% full hash table with one match per key needs to access ≈ 1.5 buckets on average to find the match using linear probing. In such a case, comparing one input key against multiple table keys cannot yield high improvement and takes no advantage of the increasing SIMD register size.

In this paper, we propose a generic form of hash table vectorization termed *vertical vectorization* that can be applied to any hash table variant without altering the hash table layout. The fundamental principle is to process a different input key per vector lane. All vector lanes process different keys from the input and access different hash table locations.

The hash table variants we discuss are linear probing (Section 5.1), double hashing (Section 5.2), and cuckoo hashing (Section 5.3). For the hash function, we use multiplicative hashing, which requires two multiplications, or for 2^n buckets, one multiplication and a shift. Multiplication costs very few cycles in mainstream CPUs and is supported in SIMD.

5.1 Linear Probing

Linear probing is an open addressing scheme that, to either insert an entry or terminate the search, traverses the table linearly until an empty bucket is found. The hash table stores keys and payloads but no pointers. The scalar code for probing the hash table is shown in Algorithm 4.

Algorithm 4 Linear Probing - Probe (Scalar)

```
j ← 0                                                    ▷ output index
for i ← 0 to |S_keys| − 1 do                  ▷ outer (probing) relation
    k ← S_keys[i]
    v ← S_payloads[i]
    h ← (k · f) × ↑ |T|      ▷ "× ↑": multiply & keep upper half
    while T_keys[h] ≠ k_empty do                    ▷ until empty bucket
        if k = T_keys[h] then
            RS_R_payloads[j] ← T_payloads[h]      ▷ inner payloads
            RS_S_payloads[j] ← v                  ▷ outer payloads
            RS_keys[j] ← k                              ▷ join keys
            j ← j + 1
        end if
        h ← h + 1                                       ▷ next bucket
        if h = |T| then                          ▷ reset if last bucket
            h ← 0
        end if
    end while
end for
```

Algorithm 5 Linear Probing - Probe (Vector)

```
i, j ← 0                                  ▷ input & output indexes (scalar register)
o⃗ ← 0                                     ▷ linear probing offsets (vector register)
m ← true                                  ▷ boolean vector register
while i + W ≤ |S_keys_in| do              ▷ W: # of vector lanes
    k⃗ ←_m S_keys[i]                       ▷ selectively load input tuples
    v⃗ ←_m S_payloads[i]
    i ← i + |m|
    h⃗ ← (k⃗ · f) ×↑ |T|                    ▷ multiplicative hashing
    h⃗ ← h⃗ + o⃗                            ▷ add offsets & fix overflows
    h⃗ ← (h⃗ < |T|) ? h⃗ : (h⃗ − |T|)        ▷ "m ? x⃗ : y⃗": vector blend
    k⃗_T ← T_keys[h⃗]                       ▷ gather buckets
    v⃗_T ← T_payloads[h⃗]
    m ← k⃗_T = k⃗
    RS_keys[j] ←_m k⃗                      ▷ selectively store matching tuples
    RS_S_payloads[j] ←_m v⃗
    RS_R_payloads[j] ←_m v⃗_T
    j ← j + |m|
    m ← k⃗_T = k_empty                     ▷ discard finished tuples
    o⃗ ← m ? 0 : (o⃗ + 1)                   ▷ increment or reset offsets
end while
```

The vectorized implementation of probing a hash table using a linear probing scheme is shown in Algorithm 5. Our vectorization principle is to process a different key per SIMD lane using gathers to access the hash table. Assuming W vector lanes, we process W different input keys on each loop. Instead of using a nested loop to find all matches for the W keys before loading the next W keys, we reuse vector lanes as soon as we know there are no more matches in the table, by selectively loading new keys from the input to replace finished keys. Thus, each key executes the same number of loops as in scalar code. Every time a match is found, we use selective stores to write to the output the vector lanes that have matches. In order to support each key having executed an arbitrary number of loops already, we keep a vector of offsets that maintain how far each key has searched in the table. When a key is overwritten, the offset is reset to zero.

A simpler approach is to process W keys at a time and use a nested loop to find all matches. However, the inner loop would be executed as many times as the maximum number of buckets accessed by any one of the W keys, underutilizing the SIMD lanes, because the average number of accessed buckets of W keys can be significantly smaller than the maximum. By reusing vector lanes dynamically, we are reading the probing input "out-of-order". Thus, the probing algorithm is no longer *stable*, i.e., the order of the output does not always match the previous order of the probing input.

Building a linear probing table is similar. We need to reach an empty bucket to insert a new tuple. The scalar code is shown in Algorithm 6 and the vector code in Algorithm 7.

Algorithm 6 Linear Probing - Build (Scalar)

```
for i ← 0 to |R_keys| − 1 do              ▷ inner (building) relation
    k ← R_keys[i]
    h ← (k · f) ×↑ |T|                    ▷ multiplicative hashing
    while T_keys[h] ≠ k_empty do          ▷ until empty bucket
        h ← h + 1                         ▷ next bucket
        if h = |T| then
            h ← 0                         ▷ reset if last
        end if
    end while
    T_keys[h] ← k                         ▷ set empty bucket
    T_payloads[h] ← R_payloads[i]
end for
```

Algorithm 7 Linear Probing - Build (Vector)

```
l⃗ ← {1, 2, 3, ..., W}                    ▷ any vector with unique values per lane
i, j ← 0 , m ← true                       ▷ input & output index & bitmask
o⃗ ← 0                                     ▷ linear probing offset
while i + W ≤ |R_keys| do
    k⃗ ←_m R_keys[i]                       ▷ selectively load input tuples
    v⃗ ←_m R_payloads[i]
    i ← i + |m|
    h⃗ ← o⃗ + (k · f) ×↑ |T|                ▷ multiplicative hashing
    h⃗ ← (h⃗ < |T|) ? h⃗ : (h⃗ − |T|)        ▷ fix overflows
    k⃗_T ← T_keys[h⃗]                       ▷ gather buckets
    m ← k⃗_T = k_empty                     ▷ find empty buckets
    T[h⃗] ←_m l⃗                           ▷ detect conflicts
    l⃗_back ←_m T_keys[h⃗]
    m ← m & (l⃗ = l⃗_back)
    T_keys[h⃗] ←_m k⃗                       ▷ scatter to buckets ...
    T_payloads[h⃗] ←_m v⃗                   ▷ ... if not conflicting
    o⃗ ← m ? 0 : (o⃗ + 1)                   ▷ increment or reset offsets
end while
```

The basics of vectorized probe and build of linear probing hash tables are the same. We process different input keys per SIMD lane and on top of gathers, we now also use scatters to store the keys non-contiguously. We access the input "out-of-order" to reuse lanes as soon as keys are inserted. To insert tuples, we first gather to check if the buckets are empty and then scatter the tuples only if the bucket is empty. The tuples that accessed a non-empty bucket increment an offset vector in order to search the next bucket in the next loop.

In order to ensure that multiple tuples will not try to fill the same empty bucket, we add a conflict detection step before scattering the tuples. Two lanes are conflicting if they point to the same location. However, we do not need to identify both lanes but rather the leftmost one that would get its value overwritten by the rightmost during the scatter. To identify these lanes, we scatter arbitrary values using a vector with unique values per lane (e.g., [1,2,3,...,W]). Then, we gather using the same index vector. If the scattered matches the gather value, the lane can scatter safely. The conflicting lanes search the next bucket in the next loop.

Future SIMD instruction sets include special instructions that can support this functionality (`vpconflictd` in AVX 3), thus saving the need for the extra scatter and gather to detect conflicts. Nevertheless, these instructions are not supported on mainstream CPUs or the Xeon Phi as of yet.

If the input keys are unique (e.g., join on a candidate key), we can scatter the keys to the table and gather them back to find the conflicting lanes instead of a constant vector with unique values per lane. Thus, we save one scatter operation.

The algorithmic descriptions show the keys and values of the hash table on separate arrays. In practice, the hash table uses an interleaved key-value layout. To halve the number of cache accesses, we pack multiple gathers into fewer wider gathers. For example, when using 32-bit keys and 32-bit payloads, the two consecutive 16-way 32-bit gathers of the above code can be replaced with two 8-way 64-bit gathers and a few shuffle operations to split keys and payloads. The same applies to scatters (see Appendix E for details).

For both probing and building, selective loads and stores assume there are enough items in the input to saturate the vector register. To process the last items in the input, we switch to scalar code. The last items are bounded in number by $2 \cdot W$, which is negligible compared to the total number of input tuples. Thus, the overall throughput is unaffected.

5.2 Double Hashing

Duplicate keys in hash tables can be handled by storing the payloads in a separate table, or by repeating the keys. The first approach works well when most matching keys are repeated. The second approach works well with mostly unique keys, but suffers from clustering duplicate keys in the same region, if linear probing is used. Double hashing uses a second hash function to distribute collisions so that the number of accessed buckets is close to the number of true matches. Thus, we can use the second approach for both cases. Comparing multiple hash table layouts based on the number of repeats is out of the scope of this paper.

Algorithm 8 Double Hashing Function

$\vec{f_L} \leftarrow m\ ?\ f_1\ :\ f_2$ ▷ *pick multiplicative hash factor*
$\vec{f_H} \leftarrow m\ ?\ |T| : (|T|-1)$ ▷ *the collision bucket ...*
$\vec{h} \leftarrow m\ ?\ 0 : (\vec{h}+1)$ ▷ *... is never repeated*
$\vec{h} \leftarrow \vec{h} + ((\vec{k} \times \downarrow \vec{f_L}) \times \uparrow \vec{f_H})$ ▷ *multiplicative hashing*
$\vec{h} \leftarrow (\vec{h} < |T|)\ ?\ \vec{h} : (\vec{h} - |T|)$ ▷ *fix overflows (no modulo)*

Algorithm 8 shows the double hashing scheme that we propose. Here, m is the subset of vector lanes that have probed at least one bucket. If the primary hash function h_1 is in $[0, |T|)$, the collision hash function h_2 is in $[1, |T|)$, and $|T|$ is prime, then $h = h_1 + N \cdot h_2$ modulo $|T|$ (double hashing) *never* repeats the same bucket for $N < |T|$ collisions. To avoid the expensive modulos, we use $h - |T|$ when $h \geq |T|$.

5.3 Cuckoo Hashing

Cuckoo hashing [23] is another hashing scheme that uses multiple hash functions. and is the only hash table scheme that has been vectorized in previous work [30], as a means to allow multiple keys per bucket (horizontal vectorization). Here, we study cuckoo hashing to compare our (vertical vectorization) approach against previous work [30, 42]. We also show that complicated control flow logic, such as cuckoo table building, can be transformed to data flow vector logic.

The scalar code for cuckoo table probing, which we omit due to space requirements, can be written in two ways. In the simple way, we check the second bucket only if the first bucket does not match. The alternative way is to always access both buckets and blend their results using bitwise operations [42]. The latter approach eliminates branching at the expense of always accessing both buckets. Still, it has been shown to be faster than other variants on CPUs [42].

Algorithm 9 Cuckoo Hashing - Probing

$j \leftarrow 0$
for $i \leftarrow 0$ **to** $|S|-1$ **step** W **do**
  $\vec{k} \leftarrow S_{keys}[i]$ ▷ *load input tuples*
  $\vec{v} \leftarrow S_{payloads}[i]$
  $\vec{h_1} \leftarrow (\vec{k} \cdot f_1) \times \uparrow |T|$ ▷ *1st hash function*
  $\vec{h_2} \leftarrow (\vec{k} \cdot f_2) \times \uparrow |T|$ ▷ *2nd hash function*
  $\vec{k_T} \leftarrow T_{keys}[\vec{h_1}]$ ▷ *gather 1st function bucket*
  $\vec{v_T} \leftarrow T_{payloads}[\vec{h_1}]$
  $m \leftarrow \vec{k} \neq \vec{k_T}$
  $\vec{k_T} \leftarrow_m T_{keys}[\vec{h_2}]$ ▷ *gather 2nd function bucket ...*
  $\vec{v_T} \leftarrow_m T_{payloads}[\vec{h_2}]$ ▷ *... if 1st is not matching*
  $m \leftarrow \vec{k} = \vec{k_T}$
  $RS_{keys}[j] \leftarrow_m \vec{k}$ ▷ *selectively store matches*
  $RS_{S\_payloads}[j] \leftarrow_m \vec{v}$
  $RS_{R\_payloads}[j] \leftarrow_m \vec{v_T}$
  $j \leftarrow j + |m|$
end for

Vectorized cuckoo table probing is shown in Algorithm 9. No inner loop is required since we have only two choices. We load W keys with an aligned vector load and gather the first bucket per key. For the keys that do not match, we gather the second bucket. Cuckoo tables do not directly support key repeats. Probing is stable by reading the input "in-order", but accesses remote buckets when out of the cache.

Building a cuckoo hashing table is more complicated. If both bucket choices are not empty, we create space by displacing the tuple of one bucket to its alternate location. This process may be repeated until an empty bucket is reached.

Algorithm 10 Cuckoo Hashing - Building

$i, j \leftarrow 0$, $m \leftarrow$ true
while $i + W \leq |R|$ **do**
  $\vec{k} \leftarrow_m R_{keys\_in}[i]$ ▷ *selectively load new ...*
  $\vec{v} \leftarrow_m R_{payloads\_in}[i]$ ▷ *... tuples from the input*
  $i \leftarrow i + |m|$
  $\vec{h_1} \leftarrow (\vec{k} \cdot f_1) \times \uparrow |B|$ ▷ *1st hash function*
  $\vec{h_2} \leftarrow (\vec{k} \cdot f_2) \times \uparrow |B|$ ▷ *2nd hash function*
  $\vec{h} \leftarrow \vec{h_1} + \vec{h_2} - \vec{h}$ ▷ *use other function if old*
  $\vec{h} \leftarrow m\ ?\ \vec{h_1} : \vec{h}$ ▷ *use 1st function if new*
  $\vec{k_T} \leftarrow T_{keys}[\vec{h}]$ ▷ *gather buckets for ...*
  $\vec{v_T} \leftarrow T_{payloads}[\vec{h}]$ ▷ *... new & old tuples*
  $m \leftarrow m\ \&\ (\vec{k_T} \neq k_{empty})$ ▷ *use 2nd function if new ...*
  $\vec{h} \leftarrow m\ ?\ \vec{h_2} : \vec{h}$ ▷ *... & 1st is non-matching*
  $\vec{k_T} \leftarrow_m T_{keys}[\vec{h}]$ ▷ *selectively (re)gather ...*
  $\vec{v_T} \leftarrow_m T_{payloads}[\vec{h}]$ ▷ *... for new using 2nd*
  $T_{keys}[\vec{h}] \leftarrow \vec{k}$ ▷ *scatter all tuples ...*
  $T_{payloads}[\vec{h}] \leftarrow \vec{v}$ ▷ *... to store or swap*
  $\vec{k_{back}} \leftarrow T_{keys}[\vec{h}]$ ▷ *gather (unique) keys ...*
  $m \leftarrow \vec{k} \neq \vec{k_{back}}$ ▷ *... to detect conflicts*
  $\vec{k} \leftarrow m\ ?\ \vec{k_T} : \vec{k}$ ▷ *conflicting tuples are ...*
  $\vec{v} \leftarrow m\ ?\ \vec{v_T} : \vec{v}$ ▷ *... kept to be (re)inserted*
  $m \leftarrow \vec{k} = k_{empty}$ ▷ *inserted tuples are replaced*
end while

Vectorized cuckoo table building, shown in Algorithm 10, reuses vector lanes to load new tuples from the input. The remaining lanes are either previously conflicting or displaced tuples. The newly loaded tuples gather buckets using one or both hash functions to find an empty bucket. The tuples that were carried from the previous loop use the alternative hash function compared to the previous loop. We scatter the tuples to the hash table and gather back the keys to detect conflicts. The lanes with newly displaced tuples, which were gathered earlier in this loop, and the conflicting lanes are passed through to the next loop. The other lanes are reused.

6. BLOOM FILTERS

Bloom filters are an essential data structure for applying selective conditions across tables before joining them, a *semi join*. A tuple qualifies from the Bloom filter, if k specific bits are set in the filter, based on k hash functions. Aborting a tuple as soon as one bit-test fails is essential to achieve high performance, because most tuples fail after a few bit tests.

Vectorized Bloom filter probing was recently shown to get a significant performance boost over scalar code on the latest mainstream CPUs, especially when the Bloom filter is cache resident [27]. The design and implementation follows the principle of processing different input keys per lane and is one of our influences for this paper. However, no fundamental vector opeations were explicitly defined. Here, we evaluate the vectorized Bloom filter design [27] on Xeon Phi.

7. PARTITIONING

Partitioning is a ubiquitous operation for modern hardware query execution as a way to split large inputs into cache-conscious non-overlapping sub-problems. For example, join and aggregation operators can use hash partitioning to split the input into small partitions that are distributed among threads and now fit in the cache [3, 4, 5, 14, 19, 26]. We study all types of partitioning: radix, hash, and range.

7.1 Radix & Hash Histogram

Prior to moving any data, in order to partition into contiguous segments, we use a histogram to set the boundaries. To compute the histogram, we increment a count based on the partition function of each key. By using multiplicative hashing, hash partitioning becomes equally fast to radix.

Algorithm 11 Radix Partitioning - Histogram

```
o⃗ ← {0, 1, 2, 3, ..., W − 1}
H_partial[P × W] ← 0                    ▷ initialize replicated histograms
for i ← 0 to |T_keys_in| − 1 step W do
    k⃗ ← T_keys_in[i]
    h⃗ ← (k⃗ << b_L) >> b_R                          ▷ radix function
    h⃗ ← o⃗ + (h⃗ · W)                    ▷ index for multiple histograms
    c⃗ ← H_partial[h⃗]                          ▷ increment W counts
    H_partial[h⃗] ← c⃗ + 1
end for
for i ← 0 to P − 1 do
    c⃗ ← H_partial[i · W]                  ▷ load W counts of partition
    H[i] ← sum_across(c⃗)                    ▷ reduce into single result
end for
```

Vectorized histogram generation, shown in Algorithm 11, uses gathers and scatters to increment counts. However, if multiple lanes scatter to the same histogram count, the count will still be incremented by 1 and all items (over)written to the same location. To avoid conflicts, we replicate the histogram to isolate each lane. Thus, lane j increments $H'[i \cdot W + j]$ instead of $H[i]$. In the end, the W histograms are reduced into one. If the histograms do not fit in the fastest cache, we use 1-byte counts and flush on overflow.

7.2 Range Histogram

Radix and hash partitioning functions are significantly faster than range partitioning functions. In range function, we execute a binary search over a sorted array of splitters. Although the array is cache resident, the number of accesses is logarithmic and all accesses are dependent on each other, thus the cache hit latency in the critical path is exposed [26]. Branch elimination only marginally improves performance.

Binary search can be vectorized using gather instructions to load the splitters from the sorted array, as shown in Algorithm 12, by processing W keys in parallel. The search path is computed by blending low and high pointers. We can assume without loss of generality that $P = 2^n$, since we can always patch the splitter array with maximum values.

Algorithm 12 Range Partitioning Function

```
l⃗ ← 0 , h⃗ ← P                          ▷ l⃗ is also the output vector
for i ← 0 to log P − 1 do
    a⃗ ← (l⃗ + h⃗) >> 1                              ▷ compute middle
    d⃗ ← D[a⃗ − 1]                                ▷ gather splitters
    m ← k⃗ > d⃗                                ▷ compare with splitters
    l⃗ ← m ? a⃗ : l⃗                              ▷ select upper half
    h⃗ ← m ? h⃗ : a⃗                              ▷ select lower half
end for
```

Recently, a range index was proposed where each node has multiple splitters that are compared against one input key using SIMD comparisons [26]. Each node is at least as wide as a vector and scalar code is used for index arithmetic and to access the nodes (without gathers), relying on the superscalar pipeline to hide the cost of scalar instructions. The SIMD index can be seen as horizontal vectorization for binary search and is evaluated on simple and complex cores.

7.3 Shuffling

The data shuffling phase of partitioning involves the actual movement of tuples. To generate the output partitions in contiguous space, we maintain an array of partition offsets, initialized by the prefix sum of the histogram. The offset array is updated for every tuple transferred to the output.

Vectorized shuffling uses gathers and scatters to increment the offset array and scatters the tuples to the output. However, if multiple vector lanes have tuples that belong to the same partition, the offset would be incremented by one and these tuples would be (over)written to the same location.

We compute a vector of conflict offsets, by using gathers and scatters to detect conflicts iteratively, as shown in Algorithm 13. First, we scatter unique values per lane to an array with P entries. Then, we gather using the same indexes and compare against the scattered vector to find conflicts. We increment the conflicting lanes and repeat the process for these lanes only until no lanes conflict. Even if W iterations are executed, the total number of accesses to distinct memory locations is always W, i.e., if a_i is the number of accesses to distinct memory locations in iteration i, then $\sum a_i = W$.

Algorithm 13 Conflict Serialization Function ($\vec{h}$, A)

```
l⃗ ← {W − 1, W − 2, W − 3, ..., 0}                ▷ reversing mask
h⃗ ← permute(h⃗, l⃗)                               ▷ reverse hashes
c⃗ ← 0 , m ← true          ▷ serialization offsets & conflict mask
repeat
    A[h⃗] ←_m l⃗                                  ▷ detect conflicts
    l⃗_back ←_m A[h⃗]
    m ← m & (l⃗ ≠ l⃗_back)                    ▷ update conflicting lanes
    c⃗ ← m ? (c⃗ + 1) : c⃗                      ▷ increment offsets ...
until m = false                              ▷ ... for conflicting lanes
return permute(c⃗, l⃗)                          ▷ reverse to original order
```

Since the rightmost lane is written during conflicts, tuples of the same partition in the same vector are written in reverse order. Also, per group of k conflicting lanes, the rightmost lane will incorrectly increment the offset by 1, not by k. By reversing the index vector during serialization, we update the offsets correctly and also maintain the input order. *Stable* partitioning is essential for algorithms such as LSB radixsort. Vectorized shuffling is shown in Algorithm 14.

Algorithm 14 Radix Partitioning - Shuffling

```
O ← prefix_sum(H)                  ▷ partition offsets from histogram
for i ← 0 to |T_keys_in| − 1 step W do
    k⃗ ← T_keys_in[i]                              ▷ load input tuples
    v⃗ ← T_payloads_in[i]
    h⃗ ← (k⃗ << b_L) >> b_R                          ▷ radix function
    o⃗ ← O[h⃗]                                  ▷ gather partition offsets
    c⃗ ← serialize_conflicts(h⃗, O)                ▷ serialize conflicts
    o⃗ ← o⃗ + c⃗                                ▷ add serialization offsets
    O[h⃗] ← o⃗ + 1                          ▷ scatter incremented offsets
    T_keys_out[o⃗] ← k⃗                              ▷ scatter tuples
    T_payloads_out[o⃗] ← v⃗
end for
```

7.4 Buffered Shuffling

Shuffling, as described so far, is fast if the input is cache resident, but falls into certain performance pitfalls when larger than the cache. First, it suffers from TLB thrashing when the partitioning fanout exceeds the TLB capacity [20]. Second, it generates many cache conflicts [31] and in the worst case, may be bound by the size of the cache associativity set. Third, using normal stores, we trigger cache loads to execute the stores and reduce the bandwidth due to loading cache lines that will only be overwritten [38].

The vectorized implementation of simple non-buffered shuffling improves performance, but suffers from the same performance pitfalls as the scalar version. In general, vectorization improves performance compared to its scalar counterpart, but does not overcome algorithmic inefficiencies.

To solve these problems, recent work proposed keeping the data in buffers and flushing them in groups [31]. If the buffers are small and packed together, they will not cause TLB or cache misses. Thus, with W buffer slots per partition, we reduce cache and TLB misses to $1/W$. If the buffers are flushed with non-temporal stores, we facilitate hardware write combining and avoid polluting the cache with output data [38]. The fanout is bounded by the cache capacity to keep the buffer cache resident. The scalar code for buffered shuffling is thoroughly described in previous work [4, 26].

The improvement of vectorized buffered shuffling shown in Algorithm 15 over vectorized unbuffered shuffling shown in Algorithm 14, is scattering the tuples to the cache resident buffer rather than directly to the output. For each vector of tuples, once the tuples are scattered, we iterate over the partitions that the current W input tuples belong to, and flush to the output when all available buffer slots are filled.

Algorithm 15 Radix Partitioning - Buffered Shuffling

$O \leftarrow$ prefix_sum(H) ▷ *partition offsets from histogram*
for $i \leftarrow 0$ **to** $|T_{keys\_in}| - 1$ **step** W **do**
  $\vec{k} \leftarrow T_{keys\_in}[i]$ ▷ *load input tuples*
  $\vec{v} \leftarrow T_{payloads\_in}[i]$
  $\vec{h} \leftarrow (\vec{k} << b_L) >> b_R$ ▷ *radix function*
  $\vec{o} \leftarrow O[\vec{h}]$ ▷ *gather partition offsets*
  $\vec{c} \leftarrow$ serialize_conflicts$(\vec{h}, O)$ ▷ *serialize conflicts*
  $\vec{o} \leftarrow \vec{o} + \vec{c}$ ▷ *add serialization offsets*
  $O[\vec{h}] \leftarrow \vec{o} + 1$ ▷ *scatter incremented offsets*
  $\vec{o}_B \leftarrow \vec{o}$ & $(W-1)$ ▷ *buffer offsets in partition*
  $m \leftarrow \vec{o}_B < W$ ▷ *find non-overflowing lanes*
  $m' \leftarrow !m$
  $\vec{o}_B \leftarrow \vec{o}_B + (\vec{h} \cdot W)$ ▷ *buffer offsets across partitions*
  $B_{keys}[\vec{o}_B] \leftarrow_m \vec{k}$ ▷ *scatter tuples to buffer ...*
  $B_{payloads}[\vec{o}_B] \leftarrow_m \vec{v}$ ▷ *... for non-overflowing lanes*
  $m \leftarrow \vec{o}_B = (W-1)$ ▷ *find lanes to be flushed*
  if $m \neq$ false **then**
    $H[0] \leftarrow_m \vec{h}$ ▷ *pack partitions to be flushed*
    for $j \leftarrow 0$ **to** $|m| - 1$ **do**
      $h \leftarrow H[j]$
      $o \leftarrow (O[h]$ & $-W) - W$ ▷ *output location*
      $\vec{k}_B \leftarrow B_{keys}[h \cdot W]$ ▷ *load tuples from buffer*
      $\vec{v}_B \leftarrow B_{payloads}[h \cdot W]$
      $T_{keys\_out}[o] \leftarrow \vec{k}_B$ ▷ *flush tuples to output ...*
      $T_{payloads\_out}[o] \leftarrow \vec{v}_B$ ▷ *... using streaming stores*
    end for
    $B_{keys}[\vec{o}_B - W] \leftarrow_{m'} \vec{k}$ ▷ *scatter tuples to buffer ...*
    $B_{payloads}[\vec{o}_B - W] \leftarrow_{m'} \vec{v}$ ▷ *... for overflowing lanes*
  end if
end for ▷ *cleanup the buffers after the loop*

Since multiple tuples can be written to the buffer for the same partition on each loop, we identify which vector lanes will not cause overflow and scatter them selectively before flushing the buffers that are full. After the buffers are flushed, we scatter the remaining tuples to the buffer.

We identify which vector lanes point to partitions that have filled their buffers using the output index. Given that tuples in multiple vector lanes can belong to the same partition, we identify the lanes that wrote to the last partition slot and ensure that we will not flush the same data twice in the same loop. Flushing occurs "horizontally" one partition at a time, after selectively storing the partitions in the stack.

Flushing data from the buffers to the output is done by streaming stores to avoid polluting the cache with output data [38]. Note that we are streaming to multiple outputs, thus, single output buffering (Section 4) does not apply.

Hash partitioning is used to split the data into groups with non-overlapping keys and has no need to be stable. Instead of conflict serialization, we detect and process conflicting lanes during the next loop. Performance is slightly increased because very few conflicts normally occur per loop if $P > W$.

As in hash tables, if the tuples are keys and rids stored on separate arrays, we do fewer and wider scatters by interleaving the two columns before storing to the buffers.

To partition multiple columns of payloads, we can either shuffle all the columns together as a unified tuple or shuffle one column at a time. Shuffling unified tuples should optimally compile specific code for each query at run time. Otherwise, we can process one column at a time using precompiled type-specialized code. In the latter approach, which we use here, during histogram generation, we store partition destinations alongside the conflict serialization offsets in a temporary array. Thus, we avoid reloading the wider keys as well as redoing conflict serialization for each column. The temporary array must be $\log P + \log W$ bits wide per entry.

8. SORTING

Sorting is used in databases as a subproblem for join and aggregation. Sorting is also used for declustering, index building, compression, and duplicate elimination. Recent work showed that large-scale sorting is synonymous to partitioning. Radixsort and comparison sorting based on range partitioning have comparable performance, by maximizing the fanout to minimize the number of partition passes [26].

Here, we implement least-significant-bit (LSB) radixsort, which is the fastest method for 32-bit keys [26]. We do not evaluate larger keys as Xeon Phi only supports 32-bit integer arithmetic in vector code. Parallel LSB radixsort splits the input equally among threads and uses the prefix sum of the histograms from all threads to interleave the partition outputs. Histogram generation and shuffling operate shared-nothing maximizing thread parallelism. By using vectorized buffered partitioning, we also maximize data parallelism.

9. HASH JOIN

Joins are one of the most frequent operators in analytical queries that can be expensive enough to dominate query execution time. Recent work has focused on comparing main-memory equi-joins, namely sort-merge join and hash join. The former is dominated by sorting [4, 14]. In the baseline hash join, the inner relation is built into a hash table and the outer relation probes the hash table to find matches.

Partitioning can be applied to hash join forming multiple variants with different strengths and weaknesses. Here, we design three hash join variants using different degrees of partitioning that also allow for different degrees of vectorization. Because the inputs are much larger than the cache, we use buffered shuffling during partitioning (Section 7.4).

In the first variant termed *no partition*, we do not use partitioning. The building procedure builds a shared hash table across multiple threads using atomic operations. The threads are synchronized using a barrier. The read-only probing procedure then proceeds without atomics. On the other hand, building the hash table cannot be fully vectorized because atomic operations are not supported in SIMD.

In the second variant termed *min partition*, we use partitioning to eliminate the use of atomics by not building a shared hash table. We partition the inner (building) relation into T parts, T being the number of threads, and build T hash tables without sharing across threads. During probing, we pick both which hash table and which bucket to search. All parts of the algorithm can be fully vectorized, after we slightly modify the code to probe across the T hash tables.

In the third variant termed *max partition*, we partition both relations until the inner relation parts are small enough to fit in a cache-resident hash table. In our implementation, the original partitioning phase splits both relations across T threads and each part is further partitioned by a single thread in one or more passes. The resulting partitions are used to build and probe hash tables in the cache, typically the L1. All parts of the algorithm can be fully vectorized.

10. EXPERIMENTAL EVALUATION

We use three platforms throughout our evaluation. The first is a Xeon Phi co-processor based on the MIC design, the second has one Haswell CPU, and the third has four high-end Sandy Bridge CPUs. Details are shown in Table 1.

We use the Haswell CPU to compare our vectorized implementations against scalar and state-of-the-art vectorized implementations, because Haswell has 256-bit SIMD registers and supports gathers, being the closest to the 512-bit SIMD registers of Xeon Phi. However, because one CPU cannot match the scale of Xeon Phi, we use four Sandy Bridge CPUs with comparable processing power and memory bandwidth to measure aggregate performance and power efficiency.

Platform	**1 CoPU**	**1 CPU**	**4 CPUs**
Market Name	Xeon Phi	Xeon	Xeon
Market Model	7120P	E3-1275v3	E5-4620
Clock Frequency	1.238 GHz	3.5 GHz	2.2 GHz
Cores × SMT	61 × 4	4 × 2	(4 × 8) × 2
Core Architecture	P54C	Haswell	Sandy Bridge
Issue Width	2-way	8-way	6-way
Reorder Buffer	N/A	192-entry	168-entry
L1 Size / Core	32+32 KB	32+32 KB	32+32 KB
L2 Size / Core	512 KB	256 KB	256 KB
L3 Size (Total)	0	8 MB	4 × 16 MB
Memory Capacity	16 GB	32 GB	512 GB
Load Bandwidth	212 GB/s	21.8 GB/s	122 GB/s
Copy Bandwidth	80 GB/s	9.3 GB/s	38 GB/s
SIMD Width	512-bit	256-bit	128-bit
Gather & Scatter	Yes & Yes	Yes & No	No & No
Power (TDP)	300 W	84 W	4 × 130 W

Table 1: Experimental platforms

To compile for Xeon Phi, we use ICC 15 with the `-mmic` and the `-no-vec` flag to avoid automatic vectorization. To compile for mainstream CPUs, we use either ICC 15 or GCC 4.9, the fastest per case. In all cases we use `-O3` optimization. For the Haswell CPU we use `-mavx2` (AVX 2) and for the Sandy Bridge CPUs we use `-mavx` (SSE 4 with VEX). Xeon Phi runs Linux 2.6 as an embedded OS, the Haswell machine has Linux 3.13 and the Sandy Bridge machine has Linux 3.2.

Unless specified otherwise, we use all hardware threads, including SMT, to minimize load and instruction latencies. All data are synthetically generated in memory and follow the uniform distribution. Uniform data are used in the most common analytical DBMS benchmarks, such as TPC-H, and are not particularly favorable to any specific operation. In fact, previous work has shown that joins, partitioning, and sorting are faster under skew [5, 26]. Nevertheless, optimizing for skew efficiency is out of the scope of this paper.

10.1 Selection Scans

Figure 5 shows the performance of selection scans on Xeon Phi and Haswell. The input table consists of 32-bit keys and 32-bit payloads and the selective condition on the key column is $k_{min} \leq k \leq k_{max}$, as shown in Section 4, where k_{min} and k_{max} are query constants. We vary the selectivity and measure the throughput of six selection scan versions, two scalar with and without branching [29], and four vectorized using two orthogonal design choices. First, we either select the qualifying tuples by extracting one bit at a time from the bitmask, or use vector selective stores. Second, we either access both keys and payloads during predicate evaluation and buffer the qualifiers, or we load the key column only and buffer the indexes of qualifiers, which are used to dereference the actual key and payload values during buffer flushing.

On Xeon Phi, scalar code is almost an order of magnitude slower than vector code, whereas on Haswell, vector code is about twice faster. Low selectivities are faster because RAM loading is faster than copying, and more payload column accesses are skipped. Branch elimination [29] improves performance on Haswell, but decreases performance on Xeon Phi due to slow `set` instructions. On Haswell, all vector versions are almost identical by saturating the bandwidth, while the branchless scalar code catches up on 10% selectivity. On Xeon Phi, avoiding payload column accesses dominates low selectivities, while using selective stores instead of extracting one tuple at a time dominates high selectivities. On the simple cores of Xeon Phi, vectorization is essential to saturate the bandwidth. On the complex cores of mainstream CPUs, vectorization remains useful for low selectivities.

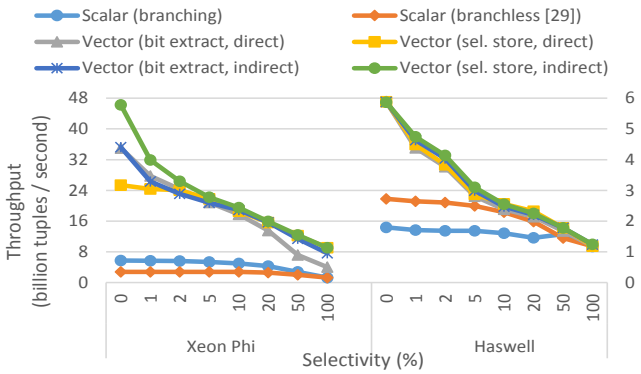

Figure 5: Selection scan (32-bit key & payload)

10.2 Hash Tables

In this section we evaluate hash table vectorization. The first set of experiments measures the probing throughput and compares against state-of-the-art scalar and vectorized approaches, on both Xeon Phi and Haswell. The second set evaluates iterative building and probing of shared-nothing hash tables on Xeon Phi only since Haswell has no scatters.

Figure 6 shows linear probing (LP) and double hashing (DH). The input is a 32-bit column with 10^9 keys, the output is a 32-bit column with the payloads of matches, almost all keys match, and the hash table is half full. The horizontal vector code uses bucketized hash table where each input key is compared against multiple table keys [30]. In our approach, vertical vector code, we compare multiple input keys against one hash table key each using gathers. We are up to 6X faster than everything else on Xeon Phi, and gain a smaller speedup for cache resident hash tables on Haswell.

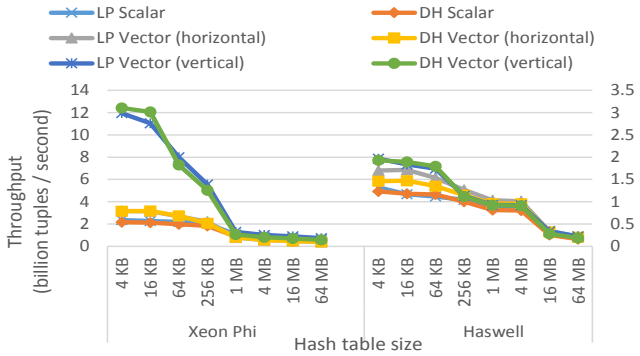

Figure 6: Probe linear probing & double hashing tables (shared, 32-bit key → 32-bit probed payload)

Figure 7 shows the probing throughput of cuckoo hashing, with the same setting as Figure 6. The scalar versions are either branching or branchless [42], and the vector versions are either horizontal (bucketized) [30] or vertical, where we evaluate two vertical techniques, loading both buckets and blending them, or loading the second bucket selectively. The branchless scalar version [42] is slower than branching scalar code on both Xeon Phi and Haswell. Also, branching scalar code is faster than branchless on Haswell using ICC, while GCC produces very slow branching code. Vertically vectorized code is 5X faster on Xeon Phi and 1.7X on Haswell.

Bucketized hash table probing is faster by using 128-bit SIMD (SSE 4) to probe 4 key, rather than use 256-bit SIMD (AVX 2) to probe 8 keys, due to faster in-register broadcasts.

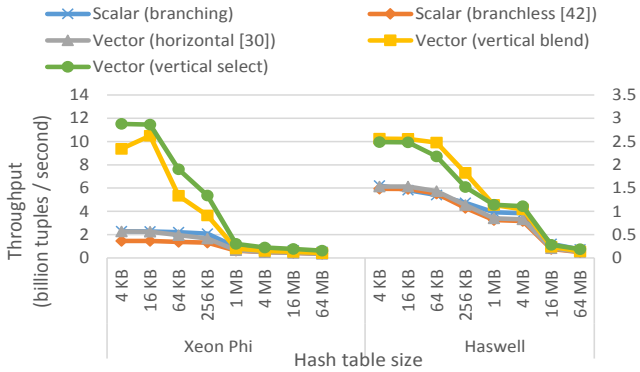

Figure 7: Probe cuckoo hashing table (2 functions, shared, 32-bit key → 32-bit probed payload)

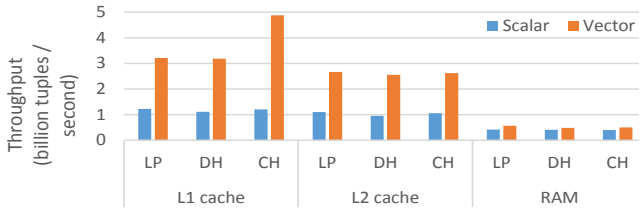

Figure 8: Build & probe linear probing, double hashing, & cuckoo hashing on Xeon Phi (1:1 build–probe, shared-nothing, 2X 32-bit key & payload)

Figure 8 interleaves building and probing of shared-nothing tables, as done in the last phase of partitioned hash join, using Xeon Phi. The build to probe ratio is 1:1, all keys match, and we vary the hash table size. The hash tables are ≈ 4 KB in L1, 64 KB in L2, and 1 MB out of cache. Both inputs have 32-bit keys and payloads, the output has the matching keys and the two payloads, the load factor is 50%, and we saturate Phi's memory. Throughput is defined as $(|R|+|S|)/t$. The speedup is 2.6–4X when the hash table resides in L1, 2.4–2.7X in L2, and 1.2–1.4X out of the cache.

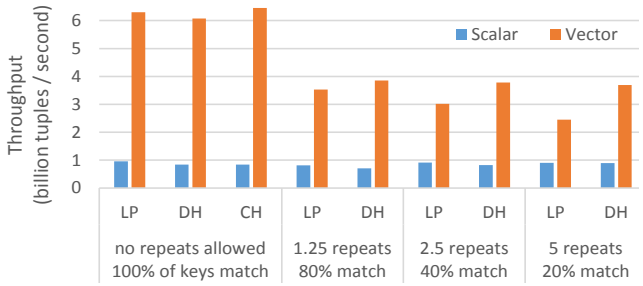

Figure 9: Build & probe linear probing, double hashing, & cuckoo hashing on Xeon Phi (1:10 build–probe, L1, shared-nothing, 2X 32-bit key & payload)

In Figure 9, we modify the experiment of Figure 8 by varying the number of key repeats with the same output size. All tables are in L1 and the build to probe ratio is 1:10. The other settings are as in Figure 8. With no key repeats, the speedup is 6.6–7.7X, higher than Figure 8, since building is more expensive than probing and also detects conflicts. Here, building alone gets a speedup of 2.5–2.7X. With 5 key repeats, the speedup is 4.1X for DH and 2.7X for LP. Thus, DH is more resilient to key repeats than LP.

10.3 Bloom Filters

In Figure 10, we measure Bloom filter probing throughput using the design of [27] with selective loads and stores for Xeon Phi. We disable loop unrolling and add buffering. The speedup is 3.6–7.8X on Xeon Phi and 1.3–3.1X on Haswell

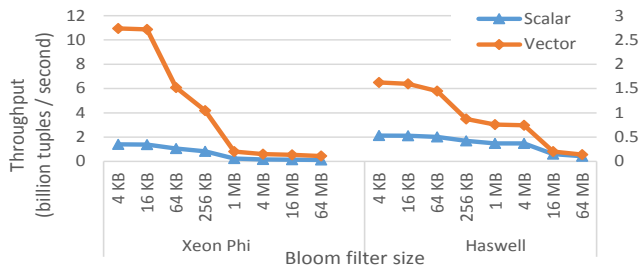

Figure 10: Bloom filter probing (5 functions, shared, 10 bits / item, 5% selectivity, 32-bit key & payload)

10.4 Partitioning

Figure 11 shows radix and hash histogram generation on Xeon Phi. On mainstream CPUs, the memory load bandwidth is saturated [26]. Scalar code is dominated by the partition function. By replicating each count W times, we get a 2.55X speedup over scalar radix. A slowdown occurs when we exceed the L1, but we can increase P by compressing to 8-bit counts. Conflict serialization does not replicate but is slower, especially if $P \leq W$. If P is too large, both count replication and conflict serialization are too expensive.

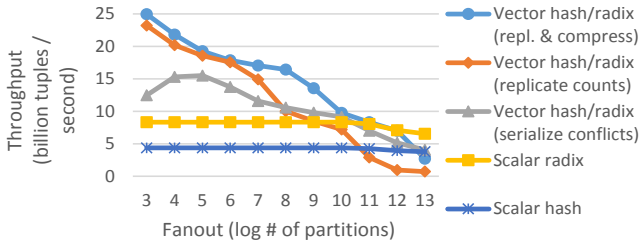

Figure 11: Radix & hash histogram on Xeon Phi

Figure 12 shows the performance of computing the range partition function. The binary search vectorization speedup is 7–15X on Xeon Phi and 2.4–2.8X on Haswell. SIMD range indexes [26] are even faster on Haswell but are slower on Xeon Phi, where the pipeline is saturated by scalar instructions. Each node has W keys and the root stays in registers.

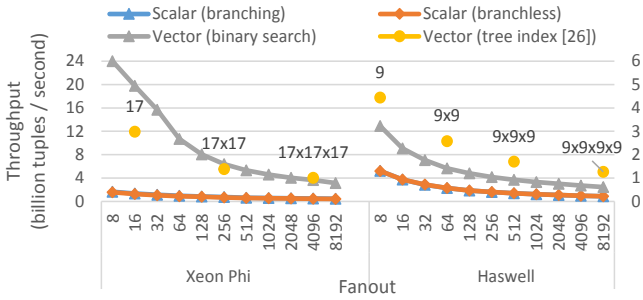

Figure 12: Range function on Xeon Phi (32-bit key)

Figure 13 measures shuffling on Xeon Phi using inputs larger than the cache. On mainstream CPUs, shuffling cannot be fully vectorized without scatters, but saturates the memory copy bandwidth at higher fanout [4, 26]. On Xeon Phi, among the unbuffered versions, vectorization achieves up to 1.95X speedup. Among the scalar versions, buffering achieves up to 1.8X speedup. Among the buffered versions, vectorization achieves up to 2.85X speedup, using up to 60% of the bandwidth. The optimal fanout, maximizing throughput × bits, is 5–8 radix bits per pass. Unstable hash partitioning, shown here, is up to 17% faster than stable radix.

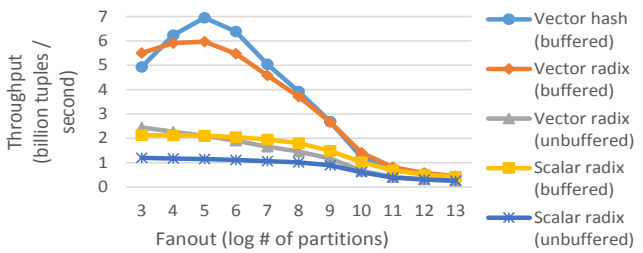

Figure 13: Radix shuffling on Xeon Phi (shared-nothing, out-of-cache, 32-bit key & payload)

10.5 Sorting & Hash Join

We now evaluate sorting and hash joins in three stages: first, we measure performance on Xeon Phi and highlight the impact of vectorization on algorithmic designs; second, we compare against 4 high-end CPUs to highlight the impact on power efficiency; and third, we study the cost of a generic implementation and materialization of multiple columns.

10.5.1 Vectorization Speedup & Algorithmic Designs

Figure 14 shows the performance of LSB radixsort on Xeon Phi. The vectorization speedup is 2.2X over state-of-the-art scalar code and time scales with size. In mainstream CPUs, we are already almost saturated, since each partitioning pass runs close to the bandwidth [31, 26, 38].

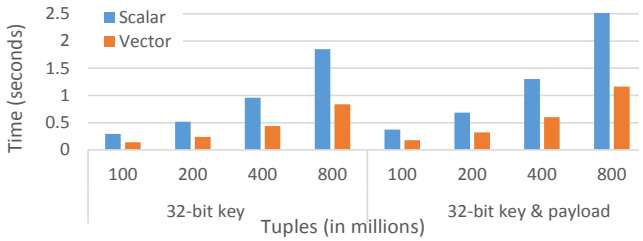

Figure 14: Radixsort on Xeon Phi (LSB)

Figure 15 shows the performance of the three hash join variants as described in Section 9, on Xeon Phi. We assume a foreign key join but our implementation is generic. The "no-partition" and the "min-partition" methods get small 1.05X and 1.25X speedups, while the fully partitioned gets a 3.3X speedup and becomes the fastest overall by a 2.25X gap, which is too large to justify hardware-oblivious joins [12]. Hash join is faster than sort-merge join [4, 14], since we sort $4 \cdot 10^8$ in 0.6 seconds and join $2 \times 2 \cdot 10^8$ tuples in 0.54 seconds.

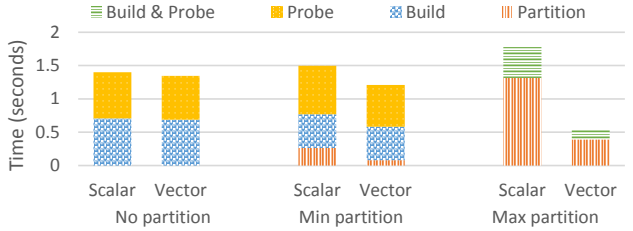

Figure 15: Multiple hash join variants on Xeon Phi ($2 \cdot 10^8 \bowtie 2 \cdot 10^8$ 32-bit key & payload)

Figure 16 shows the thread scalability of radixsort and partitioned hash join on Xeon Phi. The speedup is almost linear, even when using 2-way and 4-way SMT due to hiding load and instruction latencies. On Xeon Phi, using 4-way SMT is essential to hide the 4-cycle vector instruction latencies. On mainstream CPUs, LSB radixsort is again saturated, gaining only marginal speedup from 2-way SMT [26].

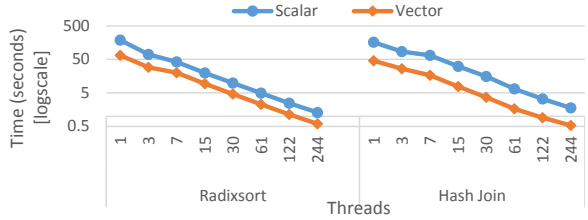

Figure 16: Radixsort & hash join scalability ($4 \cdot 10^8$ & $2 \cdot 10^8 \bowtie 2 \cdot 10^8$ 32-bit key & payload, log/log scale)

10.5.2 Aggregate Performance & Power Efficiency

We now compare Xeon Phi to 4 Sandy Bridge (SB) CPUs in order to get comparable performance, using radixsort and hash join. On the SB CPUs, we implemented partitioned hash join and use the source code of [26] for LSB radixsort.

Partitioning passes on the SB CPUs are memory bound, thus cannot benefit from full vectorization. Both radixsort and hash join on SB are NUMA aware and transfer the data at most once across CPUs [26]. To join the partitions in cache, we use horizontal linear probing, shown in Figure 6.

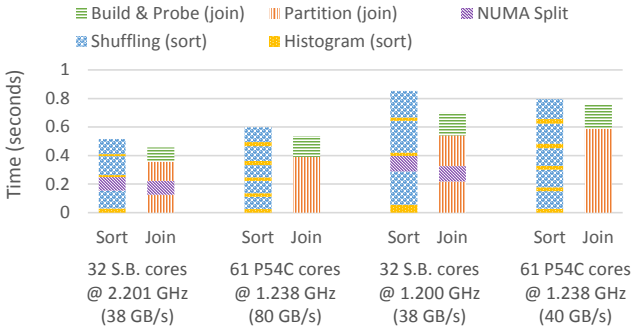

Figure 17: Radixsort & hash join on Xeon Phi 7120P versus 4 Xeon E5 4620 CPUs (sort $4 \cdot 10^8$ tuples, join $2 \cdot 10^8 \bowtie 2 \cdot 10^8$ tuples, 32-bit key & payload per table)

As shown in Figure 17, radixsort and hash join are both $\approx 14\%$ slower on the Phi compared to the 4 SB CPUs. If we assume the operating power of both platforms to be equally proportional to TDP, both radixsort and hash join are $\approx$ 1.5X more power efficient on the Phi. We also include results after equalizing the two platforms. We set the frequency of the SB CPUs to 1.2 GHz and halve the bandwidth of the Phi to 40 GB/s for copying, which is done by adjusting the code to access twice as many bytes as are processed. Phi is then 7% faster for radixsort and 8% slower for hash join.

As shown in previous work [4, 14], hash joins outperform sort-merge joins. Here, we join $2 \times 2 \cdot 10^8$ tuples faster than sorting $4 \cdot 10^8$ tuples alone, also materializing the join output.

10.5.3 Multiple Columns, Types & Materialization

Vector code cannot handle multiple types as easily as scalar code. So far we used 32-bit columns, which suffices for sorting orders and join indexes of 32-bit keys. Also, type-generic materialization methods, such as radix-decluster [21], can only do a single pass that is bounded by the cache capacity. Type-generic buffered shuffling can solve both problems.

Figure 18 measures radixsort with 32-bit keys by varying the number and width of payload columns. Per pass, we generate the histogram once and shuffle one column at a time. Shuffling 8-bit or 16-bit columns costs as much as 32-bit columns since we are compute-bound. Also, Xeon Phi upcasts 8-bit and 16-bit operations to 32-bit vector lanes. This approach scales well with wider columns, as we sort 8-byte tuples in 0.36 seconds and 36-byte tuples in 1 second.

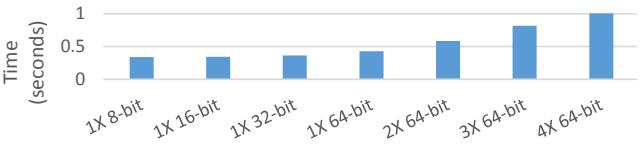

Figure 18: Radixsort with varying payloads on Xeon Phi ($2 \cdot 10^8$ tuples, 32-bit key)

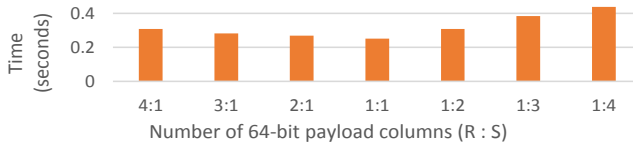

Figure 19: Hash join with varying payload columns on Xeon Phi ($10^7 \bowtie 10^8$ tuples, 32-bit keys)

Figure 19 shows partitioned hash join with 32-bit keys and multiple 64-bit payload columns. Out of the cache, we shuffle one column at a time as in Figure 18. In the cache, we store rids in hash tables and then dereference the columns.

A different materialization strategy would be, after joining keys and rids, to cluster to the order of the side with shorter payloads, and then re-partition to the order of the other side. Thus, instead of using radix-decluster [21] that is done in a single pass, we partition the shorter payloads in one or more passes to remain cache-conscious. Neverthless, finding the fastest strategy per case is out of the scope of this paper.

11. SIMD CPUS & SIMT GPUS

The SIMT model in GPUs exhibits some similarity to SIMD in CPUs. SIMT GPUs run scalar code, but "tie" all threads in a *warp* to execute the same instruction per cycle. For instance, gathers and scatters are written as scalar loads and stores to non-contiguous locations. Horizontal SIMD instructions can be supported via special shuffle instructions that operate across the warp. Thus, CPU threads are analogous to GPU warps and GPU threads are analogous to SIMD lanes. However, while conditional control flow is eliminated in SIMD code "manually", SIMT transforms control flow to data flow automatically, by executing all paths and nullifying instructions from paths that are not taken per thread.

One-to-one conversion from SIMD to SIMT code is of limited use for in-memory database operators, due to the vastly different memory hierarchy dynamics. The speedup from SIMD vectorization is maximized for cache-conscious processing, which is achieved by partitioning. On the other hand, GPUs are fast even without partitioning [13, 24], due to good memory latency hiding. Sequential operators, such as selection scans, have already been studied in detail [36].

A comparison of GPUs and Xeon Phi is out of the scope of this paper. We view Xeon Phi as a potential CPU design and study the impact of vectorization on making it more suitable for analytical databases. Thus, we do not transfer data through the PCI-e bus of Xeon Phi in our evaluation.

12. CONCLUSION

We presented generic SIMD vectorized implementations for analytical databases executing in-memory. We defined fundamental vector operations and presented good vectorization principles. We implemented selection scans, hash tables, and partitioning using entirely vector code, and then combined them to build sorting and join operators. Our implementations were evaluated against scalar and state-of-the-art vector code on the latest mainstream CPUs, as well as the Xeon Phi co-processor that is based on many simple cores with large SIMD vectors. In the context of in-memory database operators, we highlighted the impact of vectorization on algorithmic designs, as well as the architectural designs and power efficiency, by making the simple cores comparable to complex cores. Our work is applicable to all SIMD processors, using either simple or complex cores.

13. REFERENCES

[1] M.-C. Albutiu et al. Massively parallel sort-merge joins in main memory multi-core database systems. *PVLDB*, 5(10):1064–1075, June 2012.

[2] P. Bakkum et al. Accelerating SQL database operations on a GPU with CUDA. In *GPGPU*, pages 94–103, 2010.

[3] C. Balkesen et al. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*, pages 362–373, 2013.

[4] C. Balkesen et al. Multicore, main-memory joins: Sort vs. hash revisited. *PVLDB*, 7(1):85–96, Sept. 2013.

[5] S. Blanas et al. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*, pages 37–48, 2011.

[6] P. Boncz et al. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

[7] J. Chhugani et al. Efficient implementation of sorting on multi-core SIMD CPU architecture. In *VLDB*, pages 1313–1324, 2008.

[8] J. Cieslewicz et al. Automatic contention detection and amelioration for data-intensive operations. In *SIGMOD*, pages 483–494, 2010.

[9] D. J. DeWitt et al. Materialization strategies in a column-oriented DBMS. In *ICDE*, pages 466–475, 2007.

[10] J. Hofmann et al. Comparing the performance of different x86 SIMD instruction sets for a medical imaging application on modern multi- and manycore chips. *CoRR*, arXiv:1401.7494, 2014.

[11] H. Inoue et al. AA-sort: A new parallel sorting algorithm for multi-core SIMD processors. In *PACT*, pages 189–198, 2007.

[12] S. Jha et al. Improving main memory hash joins on Intel Xeon Phi processors: An experimental approach. *PVLDB*, 8(6):642–653, Feb. 2015.

[13] T. Kaldewey et al. GPU join processing revisited. In *DaMoN*, 2012.

[14] C. Kim et al. Sort vs. hash revisited: fast join implementation on modern multicore CPUs. In *VLDB*, pages 1378–1389, 2009.

[15] C. Kim et al. Fast: fast architecture sensitive tree search on modern CPUs and GPUs. In *SIGMOD*, pages 339–350, 2010.

[16] K. Krikellas et al. Generating code for holistic query evaluation. In *ICDE*, pages 613–624, 2010.

[17] S. Larsen et al. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, pages 145–156, 2000.

[18] V. Leis et al. Morsel-driven parallelism: A NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[19] S. Manegold et al. Optimizing database architecture for the new bottleneck: Memory access. *J. VLDB*, 9(3):231–246, Dec. 2000.

[20] S. Manegold et al. What happens during a join? dissecting CPU and memory optimization effects. In *VLDB*, pages 339–350, 2000.

[21] S. Manegold et al. Cache-conscious radix-decluster projections. In *VLDB*, pages 684–695, 2004.

[22] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, June 2011.

[23] R. Pagh et al. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[24] H. Pirk et al. Accelerating foreign-key joins using asymmetric memory channels. In *ADMS*, 2011.

[25] O. Polychroniou et al. High throughput heavy hitter aggregation for modern SIMD processors. In *DaMoN*, 2013.

[26] O. Polychroniou et al. A comprehensive study of main-memory partitioning and its application to large-scale comparison- and radix-sort. In *SIGMOD*, pages 755–766, 2014.

[27] O. Polychroniou et al. Vectorized Bloom filters for advanced SIMD processors. In *DaMoN*, 2014.

[28] V. Raman et al. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, Aug. 2013.

[29] K. A. Ross. Selection conditions in main memory. *ACM Trans. Database Systems*, 29(1):132–161, Mar. 2004.

[30] K. A. Ross. Efficient hash probes on modern processors. In *ICDE*, pages 1297–1301, 2007.

[31] N. Satish et al. Fast sort on CPUs and GPUs: a case for bandwidth oblivious SIMD sort. In *SIGMOD*, pages 351–362, 2010.

[32] B. Schlegel et al. Scalable frequent itemset mining on many-core processors. In *DaMoN*, 2013.

[33] L. Seiler et al. Larrabee: A many-core x86 architecture for visual computing. *ACM Trans. Graphics*, 27(3), Aug. 2008.

[34] J. Shin. Introducing control flow into vectorized code. In *PACT*, pages 280–291, 2007.

[35] L. Sidirourgos et al. Column imprints: A secondary index structure. In *SIGMOD*, pages 893–904, 2013.

[36] E. A. Sitaridi et al. Optimizing select conditions on GPUs. In *DaMoN*, 2013.

[37] M. Stonebraker et al. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[38] J. Wassenberg et al. Engineering a multi core radix sort. In *EuroPar*, pages 160–169, 2011.

[39] T. Willhalm et al. SIMD-scan: ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, Aug. 2009.

[40] Y. Ye et al. Scalable aggregation on multicore processors. In *DaMoN*, 2011.

[41] J. Zhou et al. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.

[42] M. Zukowski et al. Architecture-conscious hashing. In *DaMoN*, 2006.

APPENDIX

A. NOTATION

We now explain the notation of the vectorized algorithmic descriptions. Boolean vector operations result in boolean vector bitmasks that are shown as scalar variables. For example, $m \leftarrow \vec{x} < \vec{y}$ results in the bitmask m. Assignments such as $m \leftarrow$ true, set the W bits of m. Vectors as array indexes denote a gather or a scatter. For example, $\vec{x} \leftarrow A[y]$ is a vector load, while $\vec{x} \leftarrow A[\vec{y}]$ is a gather. Selective loads and stores as well as selective gathers and scatters use a bitmask as a subscript in the assignment. For example, $\vec{x} \leftarrow_m A[y]$ is a selective load, while $\vec{x} \leftarrow_m A[\vec{y}]$ is a selective gather. $|m|$ is the bit population count of m and $|T|$ is the size of array T. If-then-else statements, such as $\vec{x} \leftarrow m\ ?\ \vec{y} : \vec{z}$, use vector blending. Finally, scalar values in vector expressions use vectors generated before the main loop. For example, $\vec{x} \leftarrow \vec{x} + k$ and $m \leftarrow \vec{x} > k$ use a vector with k in all lanes.

B. GATHER & SCATTER

In the latest mainstream CPUs (AVX 2), gathers and scatters are executed in one instruction. On Xeon Phi, each instruction call accesses one cache line and is called repeatedly until all W lanes are processed. If N distinct cache lines are accessed, the instructions are issued N times with $N \leq W$.

An implementation for 16-way gather on Xeon Phi is shown below. The `rax` scalar register holds the array pointer. The 512-bit `zmm0` vector holds the indexes and `zmm1` vector holds the loaded values. `k1` is a 16-bit boolean vector set by `kxnor`. Each call to `vpgatherdd` finds the leftmost set bit in `k1`, extracts the index from `zmm0`, identifies other lanes pointing to the same cache line, loads the values in `zmm1`, and resets the bits in `k1`. The `jknzd` branches back if `k1` has more set bits.

```
       kxnor       %k1, %k1
loop:  vpgatherdd  (%rax, %zmm0, 4), %zmm1 {%k1}
       jknzd       loop, %k1
```

Scatters are symmetric to gathers. If multiple lanes of the index vector have the same value, the value in the rightmost lane is written but all respective bits in `k1` are reset. We can detect multiple references to the same cache line by checking if multiple bits of `k1` were reset, but we cannot distinguish between conflicts and distinct writes in the same cache line.

Non-selective gathers and scatters initially have all bits in `k1` set. Selective gathers and scatters, use `k1` and `zmm1` as both inputs and outputs to load a subset of lanes determined by `k1`. The rest lanes in `zmm1` not set in `k1` are unaffected.

The latency and throughput of each gather or scatter operation is determined by the number of cache lines accessed. Assuming all cache lines are L1 resident, the operation is faster if assembled in fewer cache lines. For the same cache lines, accessing fewer items reduces the total latency [10].

In Haswell gathers, `k1` is a vector. The asymmetry between Xeon Phi and Haswell on how gather instructions work, is possibly due to the fact that out-of-order CPUs can overlap a single high-latency branchless gather with other instructions more effectively. Haswell gathers have been shown to perform almost identically in L1 and L2. Also, loading W items from distinct cache lines in L1 or L2 has been shown to be almost as fast as loading the same item W times [10].

On Xeon Phi, mixing gather and scatter instructions in the same loop reduces performance. Thus, executing gather and scatter operations in a loop of gather and scatter instructions offers no performance benefit. Ideally, we would like to combine the spatial locality aware gathers and scatters of Xeon Phi with the single instruction of Haswell, also implementing the logic to iterate over $W' \leq W$ distinct cache lines inside the execution unit without branching.

Gathers and scatters can be emulated, if not supported. We did a careful implementation of gathers using other vector instructions and used it in linear probing and double hashing tables. We found that we lose up to 13% probing performance compared to the hardware gathers (Figure 6), which reduces as the table size increases. When the loads are not cache resident, there is no noticeable difference, which is expected since gathers and scatters go through the same memory access path as loads and stores to a single location.

C. SELECTIVE LOAD & STORE

Selective loads and stores must be able to access unaligned memory. Thus, a load can span across two cache lines. Xeon Phi provides two instructions for these two cache lines. In Haswell, we use unaligned vector accesses and permutations, which are shown in the code examples of Appendix D and E.

```
void _mm512_mask_packstore_epi32(int32_t *p, // pointer
                                 __mmask16 m, // mask
                                   __m512i v) // vector
{ _mm512_mask_packstorelo_epi32(&p[0],  m, v):
  _mm512_mask_packstorehi_epi32(&p[16], m, v); }
```

The selective load of 16 32-bit data for Xeon Phi is shown above. The symmetric selective store is shown below. The functions with prefix `_mm` are intrinsics available online.[3]

```
__m512i _mm512_mask_loadunpack_epi32(__m512i v,
                                     __mmask16 m,
                                const int32_t *p)
{ v = _mm512_mask_loadunpacklo_epi32(v, m, &p[0]):
  v = _mm512_mask_loadunpackhi_epi32(v, m, &p[16]);
  return v; }
```

[3] software.intel.com/sites/landingpage/IntrinsicsGuide/

D. SELECTION SCANS

The code for selection scans assuming 32-bit keys and payloads in Xeon Phi is shown below. We use the version that was described in Section 4 that stores input pointers of qualifiers in a cache resident buffer and then gather the data from the input. The selective condition is $k_{lower} \leq k \leq k_{upper}$. The buffer must be small enough to be L1 resident. The streaming store instruction (`_mm512_storenrngo_ps`) mechanism is to not read back the cache line that is overwritten. The input and output here must be aligned in cache lines.

```
for (i = j = k = 0;  i < tuples;  i += 16) {
  /* load key column and evaluate predicates */
  key = _mm512_load_epi32(&keys[i]);
  m = _mm512_cmpge_epi32_mask(key, mask_lower);
  m = _mm512_mask_cmple_epi32_mask(k, key, mask_upper);
  if (!_mm512_kortestz(m, m)) {    // jkzd
    /* selectively store qualifying tuple indexes */
    _mm512_mask_packstore_epi32(&rids_buf[k], m, rid);
    k += _mm_countbits_64(_mm512_mask2int(m)); }
    if (k > buf_size - 16) {
      /* flush the buffer */
      for (b = 0;  b != buf_size - 16;  b += 16) {
        ptr = _mm512_load_epi32(&rids_buf[b]);
        /* dereference column values and stream */
        key_f = _mm512_i32gather_ps(ptr, keys, 4);
        pay_f = _mm512_i32gather_ps(ptr, pays, 4);
        _mm512_storenrngo_ps(&keys_out[b + j], key_f);
        _mm512_storenrngo_ps(&pays_out[b + j], pay_f);}
      /* move extra items to the start of the buffer */
      ptr = _mm512_load_epi32(&rids_buf[b]);
      _mm512_store_epi32(&rids_buf[0], ptr);
      j += buf_size - 16;
      k -= buf_size - 16; } }
  rid = _mm512_add_epi32(rid, mask_16); }
```

We now show the same code in Haswell using AVX 2 intrinsics. The selective store is replaced by a 2-way partition that loads the permutation mask from a lookup array [27].

```
for (i = j = k = 0;  i < tuples;  i += 8) {
  /* load key columns and evaluate predicates */
  key = _mm256_load_si256((__m256i*) &keys[i]);
  cmp_lo = _mm256_cmpgt_epi32(mask_lower, key);
  cmp_hi = _mm256_cmpgt_epi32(key, mask_upper);
  cmp = _mm256_or_si256(cmp_lo, cmp_hi);
  cmp = _mm256_xor_si256(cmp, mask_minus_1);
  if (!_mm256_testz_si256(cmp, cmp)) {
    /* load permutation mask for selective store */
    m = _mm256_movemask_ps(_mm256_castsi256_ps(cmp));
    perm_comp = _mm_loadl_epi64(&perm[m]);
    perm = _mm256_cvtepi8_epi32(perm_comp);
    /* permute and store the input pointers */
    cmp = _mm256_permutevar8x32_epi32(cmp, perm);
    ptr = _mm256_permutevar8x32_epi32(rid, perm);
    _mm256_maskstore_epi32(&rids_buf[k], cmp, ptr);
    k += _mm_popcnt_u64(m);
    if (k > buf_size - 8) {
      /* flush the buffer */
      for (b = 0;  b != buf_size - 8;  b += 8) {
        /* dereference column values and store */
        ptr = _mm256_load_si256(&rids_buf[j]);
        key = _mm256_i32gather_epi32(keys, ptr, 4);
        pay = _mm256_i32gather_epi32(pays, ptr, 4);
        _mm256_stream_si256(&keys_out[b + j], key);
        _mm256_stream_si256(&pays_out[b + j], pay); }
      /* move extra items to the start of the buffer */
      ptr = _mm256_load_si256(&rids_buf[b]);
      _mm256_store_si256(&rids_buf[0], ptr);
      j += buf_size - 8;
      k -= buf_size - 8; } }
  rid = _mm256_add_epi32(rid, mask_8); }
```

E. HASH TABLES

The code for probing a hash table using linear probing for Xeon Phi is shown below. The probing input has 32-bit keys and payloads and the table stores 32-bit keys and payloads. The output includes keys and the two payload columns.

```
m = _mm512_kxnor(m, m);            // set mask
off = _mm512_xor_epi32(off, off);  // reset vector
for (i = j = 0;  i + 16 <= S_tuples;) {
  /* replace invalid keys & payloads */
  key=_mm512_mask_loadunpack_epi32(key, m, &S_keys[i]);
  pay=_mm512_mask_loadunpack_epi32(pay, m, &S_pays[i]);
  i += _mm_countbits_64(_mm512_mask2int(m));
  /* hash keys and add linear probing offset */
  hash = _mm512_mullo_epi32(key, mask_factor);
  hash = _mm512_mulhi_epu32(hash, mask_buckets);
  hash = _mm512_add_epi32(hash, off);
  /* gather key-payload pairs (buckets) */
  lo = _mm512_i32logather_epi64(hash, table, 8);
  hash =_mm512_permute4f128_epi32(hash, _MM_PERM_BADC);
  hi = _mm512_i32logather_epi64(hash, table, 8);
  /* unpack keys and payloads from pairs */
  _MM512_UNPACK_EPI32(lo, hi, table_key, table_pay);
  /* selectively store matches to output (or buffer) */
  m = _mm512_cmpeq_epi32_mask(key, table_key);
  _mm512_mask_packstore_epi32(&keys_out[j], m, key);
  _mm512_mask_packstore_epi32(&S_pays_out[j], m, pay);
  _mm512_mask_packstore_epi32(&R_pays_out[j], m,
  /* update output index */   table_pay);
  j += _mm_countbits_64(_mm512_mask2int(m));
  /* search empty buckets using special key value */
  m = _mm512_cmpeq_epi32_mask(table_key, mask_empty):
  /* increment or reset linear probing offsets */
  off = _mm512_add_epi32(off, mask_1);
  off = _mm512_mask_xor_epi32(off, m, off, off); }
```

The code for building linear probing hash tables with 32-bit keys and payloads for Xeon Phi is shown below. The implementation of double hashing is very similar with the difference being in the way the hash index is computed.

```
m = _mm512_kxnor(m, m);            // set mask
off = _mm512_xor_epi32(off, off);  // reset vector
for (i = 0;  i + 16 <= R_tuples;) {
  /* replace invalid keys & payloads */
  key=_mm512_mask_loadunpack_epi32(key, m, &R_keys[i]);
  pay=_mm512_mask_loadunpack_epi32(pay, m, &R_rids[i]);
  i += _mm_countbits_64(_mm512_mask2int(m));
  /* hash keys and add linear probing offsets */
  hash = _mm512_mullo_epi32(key, mask_factor);
  hash = _mm512_mulhi_epu32(hash, mask_buckets);
  hash = _mm512_add_epi32(hash, off);
  /* gather keys from buckets */
  tab = _mm512_i32gather_epi32(hash, table, 8);
  /* check if buckets are empty */
  m = _mm512_cmpeq_epi32_mask(tab, mask_empty);
  /* scatter unique values per vector lane */
  _mm512_mask_i32scatter_epi32(table, m, hash,
  /* gather back values */     mask_unique, 8);
  tab = _mm512_mask_i32gather_epi32(tab, m, hash,
  /* detect non-conflicting */      table, 8);
  m=_mm512_mask_cmpeq_epi32_mask(m, tab, mask_unique);
  /* packs keys and payloads in pairs */
  _MM512_PACK_EPI32(key, pay, lo, hi);
  /* scatter key-payload pairs 1-8 */
  _mm512_mask_i32loscatter_epi32(table, m, hash,
  /* scatter key-payload pairs 9-16 */  lo, 8);
  hash =_mm512_permute4f128_epi32(hash, _MM_PERM_BADC);
  mt = _mm512_kmerge2l1h(m, m);
  _mm512_mask_i32loscatter_epi32(table, mt, hash,
  /* increment or reset offsets */      hi, 8);
  off = _mm512_add_epi32(off, mask_1);
  off = _mm512_mask_xor_epi32(off, m, off, off); }
```

The constants used in the procedures are the multiplicative hashing factor (`mask_factor`), the number of hash table buckets (`mask_buckets`), a special key for empty buckets (`mask_empty`), and constant numbers. The tuples are stored in the table using an interleaved layout, thus, we access the hash table buckets using 64-bit gathers. The code used to unpack the 32-bit keys and payloads from registers with 64-bit pairs (`_MM512_UNPACK_EPI32`) as well as the code for the inverse packing (`_MM512_PACK_EPI32`), are omitted.

Conflict detection, used here for hash table building, is the most common problem in vectorization with scatters. The future AVX 3 ISA provides specialized instructions (e.g., `_mm512_conflict_epi32`) that compare all pairs of vector lanes i, j for $i < j$ and generate a bitmask per lane of vector lanes to the left with the same value. The zero lanes are non-conflicting and thus we avoid using gathers and scatters.

Hash table probing may or may not reuse the output directly. In such a case, we must use the same buffering technique we used for selection scans. The difference from the code shown is that selective stores are used to write the data to the buffer, which is then flushed with streaming stores.

We now show the same hash table probing code for Haswell, using both selective loads and stores through permutation.

```
inv = _mm256_cmpeq_epi32(inv, inv);   // set mask
off = _mm256_xor_si256(off, off);     // reset vector
for (i = j = 0;  i + 8 <= S_tuples;) {
  /* load new items and skip reloads */
  new_key = _mm256_maskload_epi32(&S_keys[i], inv);
  new_pay = _mm256_maskload_epi32(&S_vals[i], inv);
  key = _mm256_andnot_si256(inv, key);
  pay = _mm256_andnot_si256(inv, pay);
  key = _mm256_or_si256(key, new_key);
  pay = _mm256_or_si256(pay, new_pay);
  /* compute the hash function and add offsets */
  hash = _mm256_mullo_epi32(key, mask_factor);
  off = _mm256_add_epi32(off, mask_1);
  off = _mm256_andnot_si256(inv, off);
  hash = _mm256_mulhi_epu32(hash, mask_buckets);
  hash = _mm256_add_epi32(hash, off);
  /* gather data from table and unpack */
  lo = _mm256_i32gather_epi64(table_64, hash, 8);
  hash = _mm256_permute4x64_epi64(hash, 14);
  hi = _mm256_i32gather_epi64(table_64, hash, 8);
  _MM256_UNPACK_EPI32(lo, hi, table_key, table_val);
  /* check who qualifies and who is invalid */
  inv = _mm256_cmpeq_epi32(table_key, mask_empty);
  out = _mm256_cmpeq_epi32(table_key, key);
  /* load permutation masks */
  m_out = _mm256_movemask_ps(_mm256_castsi256_ps(out));
  m_inv = _mm256_movemask_ps(_mm256_castsi256_ps(inv));
  perm_out_comp = _mm_loadl_epi64(&perm[m_out]);
  perm_inv_comp = _mm_loadl_epi64(&perm[m_inv ^ 255]);
  perm_out = _mm256_cvtepi8_epi32(perm_out_comp);
  perm_inv = _mm256_cvtepi8_epi32(perm_inv_comp);
  /* permute matching items */
  out    = _mm256_permutevar8x32_epi32(out, perm_out);
  RS_key = _mm256_permutevar8x32_epi32(key, perm_out);
  S_pay  = _mm256_permutevar8x32_epi32(pay, perm_out);
  R_pay  = _mm256_permutevar8x32_epi32(table_val,
  /* store matching items */            perm_out);
  _mm256_maskstore_epi32(  &keys_out[j], out, RS_key);
  _mm256_maskstore_epi32(&S_pays_out[j], out, S_pay);
  _mm256_maskstore_epi32(&R_pays_out[j], out, R_pay);
  j += _mm_popcnt_u64(m_out);
  /* permute invalid items */
  inv = _mm256_permutevar8x32_epi32(inv, perm_inv);
  key = _mm256_permutevar8x32_epi32(key, perm_inv);
  off = _mm256_permutevar8x32_epi32(off, perm_inv);
  i += _mm_popcnt_u64(m_inv); }
```

F. PARTITIONING

We show the code for radix histogram generation using 32-bit keys for Xeon Phi below. For the radix function, we use two shifts (`mask_shift_left`, `mask_shift_right`). To compute the replicated histogram index, we multiply by the vector lanes (`mask_16`) and add the lane offset (`mask_lane`).

```
for (i = 0;  i < tuples;  i += 16) {
  /* load keys and compute the radix */
  key = _mm512_load_epi32(&keys[i]);
  part = _mm512_sllv_epi32(key, mask_shift_left);
  part = _mm512_srlv_epi32(part, mask_shift_right);
  /* compute locations in the replicated histograms */
  part = _mm512_fmadd_epi32(part, mask_16, mask_lanes);
  /* increment the partial histograms */
  count = _mm512_i32gather_epi32(part, hists, 4);
  count = _mm512_add_epi32(count, mask_1);
  _mm512_i32scatter_epi32(hists, part, count, 4); }
/* merge partial histograms */
for (p = 0;  p != partitions;  ++p) {
  count = _mm512_load_epi32(&hists[p << 4]);
  hist[p] = _mm512_reduce_add_epi32(count); }
```

To compress the partial histograms in cache, we use 8-bit counts and add code to handle overflows. The assembly of Xeon Phi allows us to specify modifiers to gather and scatter instructions, that load or store 8-bit and 16-bit memory locations. The operations are executed on 16 32-bit lanes.

We also show vectorized binary search of 32-bit keys for Xeon Phi, which is used to compute range functions. We can patch the array so that $P = 2^n$. Haswell code is identical.

```
  hi = mask_partitions;           // broadcast P
  lo = _mm512_xor_epi32(lo, lo);  // the output vector
  for (i = 0;  i != log_partitions;  ++i) {
    /* gather middle splitter */
    mid = _mm512_add_epi32(lo, hi);
    mid = _mm512_srli_epi32(mid, 1);
    del = _mm512_i32gather_epi32(mid, &splitters[-1],
    /* compare and update pointers */ 4);
    m = _mm512_cmpgt_epi32(key, del);
    lo = _mm512_mask_blend_epi32(m, lo, mid);
    hi = _mm512_mask_blend_epi32(m, mid, hi); }
```

The code for conflict serialization using gathers and scatters is shown below. The constant permutation mask used to reverse the lanes of a vector (`mask_reverse`), also used to detect conflicts since due to having distinct values per lane. Using the conflict detection instruction provided by AVX 3, we can also implement conflict serialization. Given that AVX 3 does not provide vectorized bit count, we run a loop that clears one bit at a time per lane using `x & (x - 1)` and increments the non-zero lanes in the resulting offset vector.

```
__m512i _mm512_serialize_conflicts(__m512i part,
  /* reverse order of indexes */   int32_t *array) {
  part = _mm512_permutevar_epi32(part, mask_reverse);
  __m512i back, res = _mm512_xor_epi32(res, res);
  __mmask16 m = _mm512_kxnor(m, m);
  do {
    /* scatter unique values per lane */
    _mm512_mask_i32scatter_epi32(array, m, part,
    /* gather back values */     mask_reverse);
    back = _mm512_mask_i32gather_epi32(back, m, part,
    /* detect conflicting lanes */     array, 4);
    m = _mm512_mask_cmpneq_epi32_mask(m, back,
    /* increment offsets */           mask_reverse);
    res = _mm512_mask_add_epi32(res, m, res, mask_1);
  } while (!_mm512_kortestz(m, m));
  /* reverse result back to original order */
  return _mm512_permutevar_epi32(res, mask_reverse); }
```

The code for stable buffered shuffling for radix partitioning using 32-bit keys and payloads for Xeon Phi is shown below. Both the input and the output are assumed to be aligned on cache line boundaries. After the loop, we clean the buffer by flushing remaining tuples. If multiple threads are used, the buffer cleanup occurs after synchronizing, to fix the first cache line of each partition that may be corrupted.

When partitioning columns of multiple widths, assuming we partition one column at a time, the number of buffered items per partition varies. We implement 8-bit, 16-bit, 32-bit, and 64-bit shuffling by storing 64, 32, 16, and 8 items in the buffer per partition respectively. When we partition 64-bit columns, because 16 64-bit values can span across two cache lines, we scatter the tuples to the buffers in three phases instead of two, as overflows can occur twice per loop.

```
for (i = 0;  i < tuples;  i += 16) {
  /* load keys and payloads */
  key = _mm512_load_epi32(&keys[i]);
  pay = _mm512_load_epi32(&pays[i]);
  /* compute partition function (radix) */
  part = _mm512_sllv_epi32(key, mask_shift_left);
  part = _mm512_srlv_epi32(part, mask_shift_right);
  /* load current partition offsets */
  off = _mm512_i32gather_epi32(part, offsets, 4);
  /* detect conflicts (pollutes offsets array) */
  ser_off = _mm512_serialize_conflicts(part, offsets);
  /* scatter updated offsets (fixes offsets array) */
  off_back = _mm512_add_epi32(off_back, mask_1);
  off_back = _mm512_add_epi32(off, ser_off);
  _mm512_i32scatter_epi32(offsets, part, off_back, 4);
  _MM512_PACK_EPI32(key, pay, lo, hi);
  /* compute partition offsets in buffers */
  off = _mm512_and_epi32(off, mask_15);
  off = _mm512_add_epi32(off, ser_off);
  off_lo = _mm512_fmadd_epi32(part, mask_16, off);
  off_hi = _mm512_permute4f128_epi32(off_lo,
  /* find non-overflowing lanes */   _MM_PERM_BCDC);
  m = _mm512_cmpgt_epi32_mask(mask_15, off);
  mt = _mm512_knot(m);
  /* scatter pairs 1-8 to buffers (non-overflowing) */
  _mm512_mask_i32loscatter_epi64(buffers, m, off_lo,
  /* scatter pairs 9-16 to buffers ... */  lo, 8);
  m = _mm512_kmerge2l1h(m, m);
  _mm512_mask_i32loscatter_epi64(buffers, m, off_hi,
  /* find lanes with partitions to flush */  hi, 8);
  m = _mm512_cmpeq_epi32_mask(off, mask_15);
  if (!_mm512_kortestz(m, m)) {
    /* pack partitions that need to be flushed */
    _mm512_mask_packstorelo_epi32(flush_part, m, part);
    /* count how many partitions to flush */
    j = 0, k = _mm_countbits_64(_mm512_mask2int(m));
    do {
      p = flush_part[j];              // which partition
      o = (offsets[p] & -16) - 16;  // output location
      /* load key-payload pairs from the buffer */
      lo_f = _mm512_load_ps(&buffers[(p << 4) + 0]);
      hi_f = _mm512_load_ps(&buffers[(p << 4) + 8]);
      /* unpack pairs into keys and payloads */
      _MM512_UNPACK_PS(lo_f, hi_f, key_f, pay_f);
      /* flush to output using streaming stores */
      _mm512_storenrngo_ps(&keys_out[o], key_f);
      _mm512_storenrngo_ps(&pays_out[o], pay_f);
    } while (++j != k);
    if (!_mm512_kortestz(mt, mt)) {
      /* scatter pairs 1-8 to buffers (overflowing) */
      _mm512_mask_i32loscatter_epi64(&buffers[-16], mt,
      /* scatter pairs 9-16 ... */   off_lo, lo, 8);
      mt = _mm512_kmerge2l1h(mt, mt);
      _mm512_mask_i32loscatter_epi64(&buffers[-16], mt,
/* flush buffers after the loop */   off_hi, hi, 8);}}}
```