RETHINKING SIMD VECTORIZATION FOR IN-MEMORY DATABASES

Orestis Polychroniou

Arun Raghavan

Kenneth A. Ross

University at Buffalo The State University of New York

University at Buffalo The State University of New York

Modern Hardware

Today's servers have large amounts of main memory

For example, AMD Epyc 7763

- 256MB of cache
- up to 4TB of DDR4-3200 of ECC Memory

Entire databases can be placed in-memory, a long way from measuring IO cost in blocks of HDDs

Novel encoding and compression schemes of column store architectures reduce need for RAM access even further

Modern Hardware

Three levels of parallelism are found in modern processors

- Thread parallelism
- Instruction-level parallelism
- Data parallelism

Mainstream CPUs feature superscalar pipelines, out-of-order execution for multiple instructions and advanced SIMD vectors, all replicated on multiple cores on the same CPU

Modern Hardware

An alternate architecture (Intel® MIC)

Remove superscalar pipeline, OOOE, L3 cache

Reduce area, power consumption of individual core and pack many of them on a single chip

Augment it with large SIMD registers, advanced SIMD instructions and simultaneous multithreading on top.

Xeon-Phi is not a GPU. It has high FLOP throughput.

Previous Work

Past attempts to make use of the SIMD architecture have included:

- Optimize sequential access operators (index, linear scan)
- Multi-way trees which mimic SIMD registers
- Problem-specific operator tweaking with ad-hoc vectorization (sorting)

FUNDAMENTAL OPERATIONS

Selection Scans,

Hash Tables,

Bloom Filters,

Partitioning

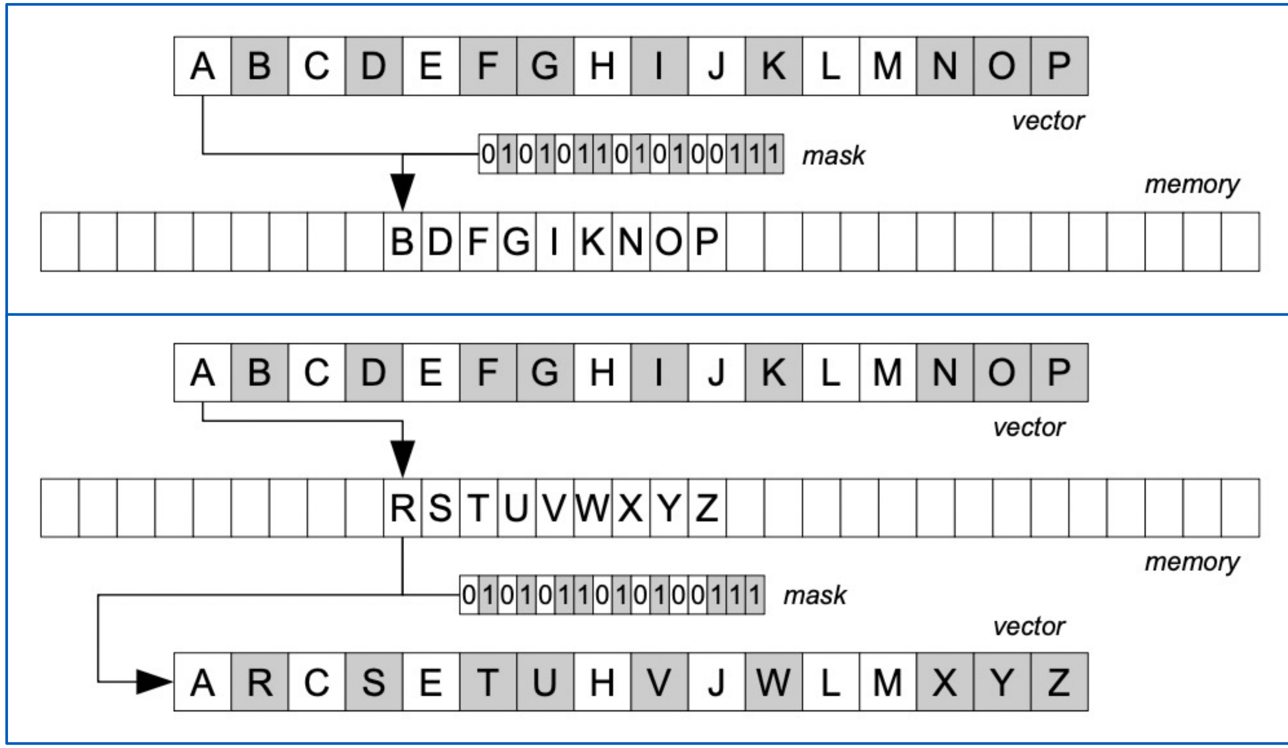
Some Primitives

- Selective Store

It takes a subset of vector lanes and stores it contiguously in memory. The subset is selected using a mask register.

- Selective Load

It takes a contiguous section of memory and writes it onto a subset of vector lanes specified by a mask. Inactive lanes retain their data.



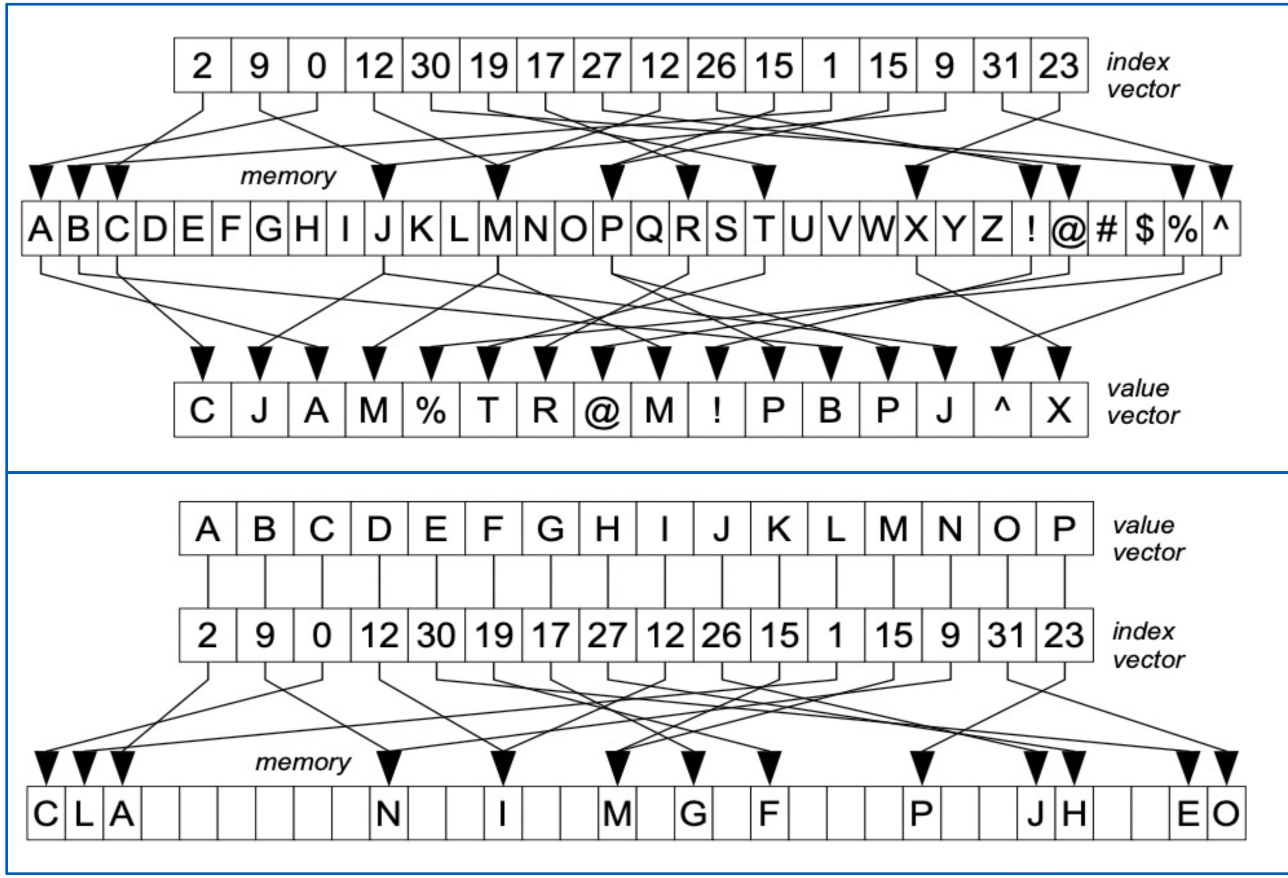
Some Primitives

- Gather

This operation loads non-contiguous data from memory using a vector of indices and a pointer.

- Scatter

This operation executes stores to various locations using the index vector and the array pointer.



Selection Scans

Selection Scans have made a comeback for main-memory query execution, with optimizations such as

- bit compression
- statistics generation
- bitmap/zone map scanning

Selection Scans

Linear selection scan with branches (Algorithm 1) can be prone to branch mispredictions. Converting control flow to data flow can affect performance, making different approaches optimal per selectivity rate.

Branchless algorithm can avoid the first penalty at the cost of accessing all payload columns and eagerly evaluating all selective predicates.

Algorithm 1 Selection Scan (Scalar - Branching)

$j \leftarrow 0$ ▷ *output index*

for $i \leftarrow 0$ **to** $|T_{keys\_in}| - 1$ **do**

  $k \leftarrow T_{keys\_in}[i]$ ▷ *access key columns*

  if $(k \geq k_{lower})$ && $(k \leq k_{upper})$ **then** ▷ *short circuit and*

    $T_{payloads\_out}[j] \leftarrow T_{payloads\_in}[i]$ ▷ *copy all columns*

    $T_{keys\_out}[j] \leftarrow k$

    $j \leftarrow j + 1$

  end if

end for

Algorithm 2 Selection Scan (Scalar - Branchless)

$j \leftarrow 0$ ▷ *output index*

for $i \leftarrow 0$ **to** $|T_{keys\_in}| - 1$ **do**

  $k \leftarrow T_{keys\_in}[i]$ ▷ *copy all columns*

  $T_{payloads\_out}[j] \leftarrow T_{payloads\_in}[i]$

  $T_{keys\_out}[j] \leftarrow k$

  $m \leftarrow (k \geq k_{lower}\ ?\ 1 : 0)\ \&\ (k \leq k_{upper}\ ?\ 1 : 0)$

  $j \leftarrow j + m$ ▷ *if-then-else expressions use conditional ...*

end for ▷ *... flags to update the index without branching*

Selection Scans

The vectorized algorithm makes use of the selective store primitive to store all the qualified tuples in the vector at once.

A small index cache of qualifiers is used instead of storing actual record values. When this buffer is full, the indexes are reloaded, and the actual columns are read and flushed to the output.

Xeon Phi provides a method like a streaming store to write a vector directly to a cache line without loading it, removing the need for the buffer write.

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
```

Hash Tables

Hash tables have uses in the execution of joins and aggregations as they allow constant time key lookups.

SIMD has been utilized to build bucketized hash tables, where a probing key can be compared to multiple hash keys by horizontal vectorization.

However, this method has diminishing results if the number of buckets to be searched is less.

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
```

Hash Tables

A generic form of vectorization is proposed, vertical vectorization, that can be applied to any hash table without modification.

The principle is to process a hash key in each vector lane. Thus, each vector lane accesses different hash table location.

This paper test three different hash table variations, linear probing, double hashing and cuckoo hashing.

The hash function used is multiplicative hashing.

Algorithm 5 Linear Probing - Probe (Vector)

$i, j \leftarrow 0$ ▷ *input & output indexes (scalar register)*
$\vec{o} \leftarrow 0$ ▷ *linear probing offsets (vector register)*
$m \leftarrow \text{true}$ ▷ *boolean vector register*
while $i + W \leq |S_{keys\_in}|$ **do** ▷ *W: # of vector lanes*
  $\vec{k} \leftarrow_m S_{keys}[i]$ ▷ *selectively load input tuples*
  $\vec{v} \leftarrow_m S_{payloads}[i]$
  $i \leftarrow i + |m|$
  $\vec{h} \leftarrow (\vec{k} \cdot f) \times \uparrow |T|$ ▷ *multiplicative hashing*
  $\vec{h} \leftarrow \vec{h} + \vec{o}$ ▷ *add offsets & fix overflows*
  $\vec{h} \leftarrow (\vec{h} < |T|)\ ?\ \vec{h} : (\vec{h} - |T|)$ ▷ *"$m\ ?\ \vec{x} : \vec{y}$": vector blend*
  $\vec{k}_T \leftarrow T_{keys}[\vec{h}]$ ▷ *gather buckets*
  $\vec{v}_T \leftarrow T_{payloads}[\vec{h}]$
  $m \leftarrow \vec{k}_T = \vec{k}$
  $RS_{keys}[j] \leftarrow_m \vec{k}$ ▷ *selectively store matching tuples*
  $RS_{S\_payloads}[j] \leftarrow_m \vec{v}$
  $RS_{R\_payloads}[j] \leftarrow_m \vec{v}_T$
  $j \leftarrow j + |m|$
  $m \leftarrow \vec{k}_T = k_{empty}$ ▷ *discard finished tuples*
  $\vec{o} \leftarrow m\ ?\ 0 : (\vec{o} + 1)$ ▷ *increment or reset offsets*
end while

Linear Probing

Linear probing is an open addressing scheme which linear traverses the hash table until an empty bucket is found, or search is terminated. Algorithm 4 shows the scalar method.

Algorithm 5 shows the vector method where the lanes are filled by a gather operation. The lanes for unmatched keys are reused by selective load to avoid the use of nested loops. The matched keys are selectively stored in memory.

An offset vector is maintained to count how far a key has searched(looped), if the key is overwritten, then the offset counter is reset.

The dynamic nature of this probing makes the algorithm unstable.

Building a linear probing table is similar.

Algorithm 5 Linear Probing - Probe (Vector)

$i, j \leftarrow 0$ ▷ *input & output indexes (scalar register)*
$\vec{o} \leftarrow 0$ ▷ *linear probing offsets (vector register)*
$m \leftarrow \text{true}$ ▷ *boolean vector register*
while $i + W \leq |S_{keys\_in}|$ **do** ▷ *W: # of vector lanes*
  $\vec{k} \leftarrow_m S_{keys}[i]$ ▷ *selectively load input tuples*
  $\vec{v} \leftarrow_m S_{payloads}[i]$
  $i \leftarrow i + |m|$
  $\vec{h} \leftarrow (\vec{k} \cdot f) \times\uparrow |T|$ ▷ *multiplicative hashing*
  $\vec{h} \leftarrow \vec{h} + \vec{o}$ ▷ *add offsets & fix overflows*
  $\vec{h} \leftarrow (\vec{h} < |T|)\ ?\ \vec{h} : (\vec{h} - |T|)$ ▷ *"$m\ ?\ \vec{x} : \vec{y}$": vector blend*
  $\vec{k}_T \leftarrow T_{keys}[\vec{h}]$ ▷ *gather buckets*
  $\vec{v}_T \leftarrow T_{payloads}[\vec{h}]$
  $m \leftarrow \vec{k}_T = \vec{k}$
  $RS_{keys}[j] \leftarrow_m \vec{k}$ ▷ *selectively store matching tuples*
  $RS_{S\_payloads}[j] \leftarrow_m \vec{v}$
  $RS_{R\_payloads}[j] \leftarrow_m \vec{v}_T$
  $j \leftarrow j + |m|$
  $m \leftarrow \vec{k}_T = k_{empty}$ ▷ *discard finished tuples*
  $\vec{o} \leftarrow m\ ?\ 0 : (\vec{o} + 1)$ ▷ *increment or reset offsets*
end while

Linear Probing

The vectorized build happens in a similar out-of-order fashion where the lanes are reused as soon as the keys are inserted. The lanes are filled and emptied with gathers to check if the bucket is empty and scatters only if the bucket is empty.

There is conflict detection step before the scatter operation to avoid clashing of keys.

- A rudimentary way is to scatter is sequential array and gather it again to check for repeats.
- AVX3 and later have a special instruction $vpconflictd$ which streamlines the conflict detection process.
- If the keys are unique then that itself can be scattered to check conflict.

Algorithm 6 Linear Probing - Build (Scalar)

for $i \leftarrow 0$ **to** $|R_{keys}| - 1$ **do** ▷ *inner (building) relation*
  $k \leftarrow R_{keys}[i]$
  $h \leftarrow (k \cdot f) \times\uparrow |T|$ ▷ *multiplicative hashing*
  while $T_{keys}[h] \neq k_{empty}$ **do** ▷ *until empty bucket*
    $h \leftarrow h + 1$ ▷ *next bucket*
    if $h = |T|$ **then**
      $h \leftarrow 0$ ▷ *reset if last*
    end if
  end while
  $T_{keys}[h] \leftarrow k$ ▷ *set empty bucket*
  $T_{payloads}[h] \leftarrow R_{payloads}[i]$
end for

Double Hashing

Double hashing is used to handle the case of duplicate keys, where linear probing would lead to collisions by clustering duplicate keys in the same region.

Double hashing distributes collision such that number of buckets accessed is close to number of true matches.

Thus, we can get away with repeating the keys.

Algorithm 8 describes the proposed function.

Algorithm 7 Linear Probing - Build (Vector)

$\vec{l} \leftarrow \{1, 2, 3, ..., W\}$ ▷ *any vector with unique values per lane*
$i, j \leftarrow 0$, $m \leftarrow \text{true}$ ▷ *input & output index & bitmask*
$\vec{o} \leftarrow 0$ ▷ *linear probing offset*
while $i + W \leq |R_{keys}|$ **do**
  $\vec{k} \leftarrow_m R_{keys}[i]$ ▷ *selectively load input tuples*
  $\vec{v} \leftarrow_m R_{payloads}[i]$
  $i \leftarrow i + |m|$
  $\vec{h} \leftarrow \vec{o} + (k \cdot f) \times\uparrow |T|$ ▷ *multiplicative hashing*
  $\vec{h} \leftarrow (\vec{h} < |T|)\ ?\ \vec{h} : (\vec{h} - |T|)$ ▷ *fix overflows*
  $\vec{k}_T \leftarrow T_{keys}[\vec{h}]$ ▷ *gather buckets*
  $m \leftarrow \vec{k}_T = k_{empty}$ ▷ *find empty buckets*
  $T[\vec{h}] \leftarrow_m \vec{l}$ ▷ *detect conflicts*
  $\vec{l}_{back} \leftarrow_m T_{keys}[\vec{h}]$
  $m \leftarrow m\ \&\ (\vec{l} = \vec{l}_{back})$
  $T_{keys}[\vec{h}] \leftarrow_m \vec{k}$ ▷ *scatter to buckets ...*
  $T_{payloads}[\vec{h}] \leftarrow_m \vec{v}$ ▷ *... if not conflicting*
  $\vec{o} \leftarrow m\ ?\ 0 : (\vec{o} + 1)$ ▷ *increment or reset offsets*
end while

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)*

Cuckoo Hashing

Cuckoo hashing allows for direct comparison with the previous horizontal vectorization solution and the proposed vertical vectorization solution.

This hashing scheme also uses multiple hash functions.

The scalar algorithm for this method can be written one of two ways:

- Check the second bracket only if the first doesn't match. This branching is prone to mis-predications.
- Check both buckets and blend the results using bitwise operations. Even with extra memory access this method is faster on CPUs.

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

Cuckoo Hashing

Algorithm 9 shows the simple vectorized probing of Cuckoo hashing.

After loading W keys, we gather the first bucket for each of those who match. For the keys that don't match we gather the second bucket.

The algorithm is stable when the input is read in order.

Vectorized Cuckoo table building is shown in Algorithm 10. Only those keys which conflict or those which were displaced after conflict check persist through the loop, the rest of the lanes are reused.

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*
  $i \leftarrow i + |m|$
  $\vec{h}_1 \leftarrow (\vec{k} \cdot f_1) \times\uparrow |B|$ ▷ *1^{st} hash function*
  $\vec{h}_2 \leftarrow (\vec{k} \cdot f_2) \times\uparrow |B|$ ▷ *2^{nd} 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 1^{st} 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 2^{nd} function if new ...*
  $\vec{h} \leftarrow m \ ? \ \vec{h}_2 : \vec{h}$ ▷ *... & 1^{st} 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 2^{nd}*
  $T_{keys}[\vec{h}] \leftarrow \vec{k}$ ▷ *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

Bloom Filters

Bloom filters are used to apply selective conditions across tables before joining them.

A record qualifies from the filter, if k specific bits are set in the filter, based on k hash functions.

Vectorized bloom filter has a great performance especially when it is cache resident. It is implemented using standard procedure and does need any vector operations to be defined.

Partitioning

Partitioning is a ubiquitous operation that splits large input into cache conscious non-overlapping sub problems.

Three types of schemes are discussed

- Radix
- Hash
- Range

Partitioning

Prior to moving any data, boundaries are set using a histogram.

Vectorized radix and hash histogram generation is shown in algorithm 11. It uses gathers and scatters to increment counts based on partition function of each key.

Even if multiple lanes scatter to the same histogram count, conflicts are avoided by isolating each lane.

Algorithm 11 Radix Partitioning - Histogram

$\vec{o} \leftarrow \{0, 1, 2, 3, ..., W-1\}$

$H_{partial}[P \times W] \leftarrow 0$ ▷ *initialize replicated histograms*

for $i \leftarrow 0$ **to** $|T_{keys\_in}| - 1$ **step** W **do**

  $\vec{k} \leftarrow T_{keys\_in}[i]$

  $\vec{h} \leftarrow (\vec{k} << b_L) >> b_R$ ▷ *radix function*

  $\vec{h} \leftarrow \vec{o} + (\vec{h} \cdot W)$ ▷ *index for multiple histograms*

  $\vec{c} \leftarrow H_{partial}[\vec{h}]$ ▷ *increment W counts*

  $H_{partial}[\vec{h}] \leftarrow \vec{c} + 1$

end for

for $i \leftarrow 0$ **to** $P - 1$ **do**

  $\vec{c} \leftarrow H_{partial}[i \cdot W]$ ▷ *load W counts of partition*

  $H[i] \leftarrow \text{sum\_across}(\vec{c})$ ▷ *reduce into single result*

end for

Partitioning

Range histogram is slower than radix and hash functions, as it uses binary search over a sorted array of splitters.

Even if array is cache-resident, the cache hit latency in the critical path is exposed.

A SIMD index is used as a horizontal vectorization for binary search to be evaluated over simple and complex cores.

Algorithm 12 Range Partitioning Function

$\vec{l} \leftarrow 0$, $\vec{h} \leftarrow P$ ▷ *$\vec{l}$ is also the output vector*
for $i \leftarrow 0$ **to** $\log P - 1$ **do**
  $\vec{a} \leftarrow (\vec{l} + \vec{h}) >> 1$ ▷ *compute middle*
  $\vec{d} \leftarrow D[\vec{a} - 1]$ ▷ *gather splitters*
  $m \leftarrow \vec{k} > \vec{d}$ ▷ *compare with splitters*
  $\vec{l} \leftarrow m\ ?\ \vec{a} : \vec{l}$ ▷ *select upper half*
  $\vec{h} \leftarrow m\ ?\ \vec{h} : \vec{a}$ ▷ *select lower half*
end for

Partitioning

The shuffling phase of partitioning involves moving the data tuples / records. The prefix sum of histograms is used as partition offsets and is updated for every tuple transferred.

Algorithm 13 handles the conflict management for the vectorized shuffling where multiple lanes might go to the same partition in the same operation.

The actual shuffling is shown in 14

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

$\vec{l} \leftarrow \{W-1, W-2, W-3, ..., 0\}$ ▷ *reversing mask*
$\vec{h} \leftarrow \text{permute}(\vec{h}, \vec{l})$ ▷ *reverse hashes*
$\vec{c} \leftarrow 0$, $m \leftarrow \text{true}$ ▷ *serialization offsets & conflict mask*
repeat
  $A[\vec{h}] \leftarrow_m \vec{l}$ ▷ *detect conflicts*
  $\vec{l}_{back} \leftarrow_m A[\vec{h}]$
  $m \leftarrow m$ & $(\vec{l} \neq \vec{l}_{back})$ ▷ *update conflicting lanes*
  $\vec{c} \leftarrow m$? $(\vec{c} + 1)$: $\vec{c}$ ▷ *increment offsets ...*
until $m = \text{false}$ ▷ *... for conflicting lanes*
return $\text{permute}(\vec{c}, \vec{l})$ ▷ *reverse to original order*

Algorithm 14 Radix Partitioning - Shuffling

$O \leftarrow \text{prefix\_sum}(H)$ ▷ *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*
  $\vec{c} \leftarrow \text{serialize\_conflicts}(\vec{h}, O)$ ▷ *serialize conflicts*
  $\vec{o} \leftarrow \vec{o} + \vec{c}$ ▷ *add serialization offsets*
  $O[\vec{h}] \leftarrow \vec{o} + 1$ ▷ *scatter incremented offsets*
  $T_{keys\_out}[\vec{o}] \leftarrow \vec{k}$ ▷ *scatter tuples*
  $T_{payloads\_out}[\vec{o}] \leftarrow \vec{v}$
end for

Partitioning

Non-buffered shuffling is great when input is cache-resident but has a host of problems when input is larger in size such as TLB thrashing, cache conflicts, and cache associativity set limitations. It is true even for the vectorized shuffling.

A proposed solution for this is to keep data in buffers and flush them in groups. Then we keep these buffers small and packed together in cache.

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*

Sorting

We use sorting largely in join and aggregation operations. They are also used for de-clustering, compression, deduplication, etc.

Large-scale sorting is shown to be synonymous to partitioning. So, we implement LSB radix sort for 32-bit keys.

Histogram generation and shuffling operate shared-nothing, maximizing thread parallelism.

By using vectorized buffered partitioning, we also maximize data parallelism.

Hash Join

Main memory equi-joins include sort-merge joins and hash joins. 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.

Three variants of hash joins are implemented.

- No partition: A shared hash table is used across threads using atomic operations. Cannot be SIMD as atomic operations are not supported.
- Min partition: Inner relation is partitioned into T(# thread) parts, creating T hash tables which are not shared. Entire algorithm can be vectorized.
- Max partition: Both inner and outer relations are partitioned such that inner partition is small enough to fit in a cache-resident hash table. Fully vectorized.

University at Buffalo The State University of New York

EXPERIMENTAL EVALUATION

Xeon Phi

Haswell Xeon

4x Sandy Bridge Xeon

Test Platform

Three platforms are used for evaluation.

- Xeon Phi co-processor based on the MIC design.
- Haswell Xeon with 256-bit SIMD registers to compare scalar and S.O.T.A. vector solutions.
- 4x Sandy Bridge Xeons to measure aggregate performance and 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 $\times$ SMT	61×4	4×2	$(4 \times 8) \times 2$
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

Selection Scans

We vary the selectivity and measure the throughput of six selection scan versions, two scalar with and without branching, and four vectorized using two orthogonal design choices.

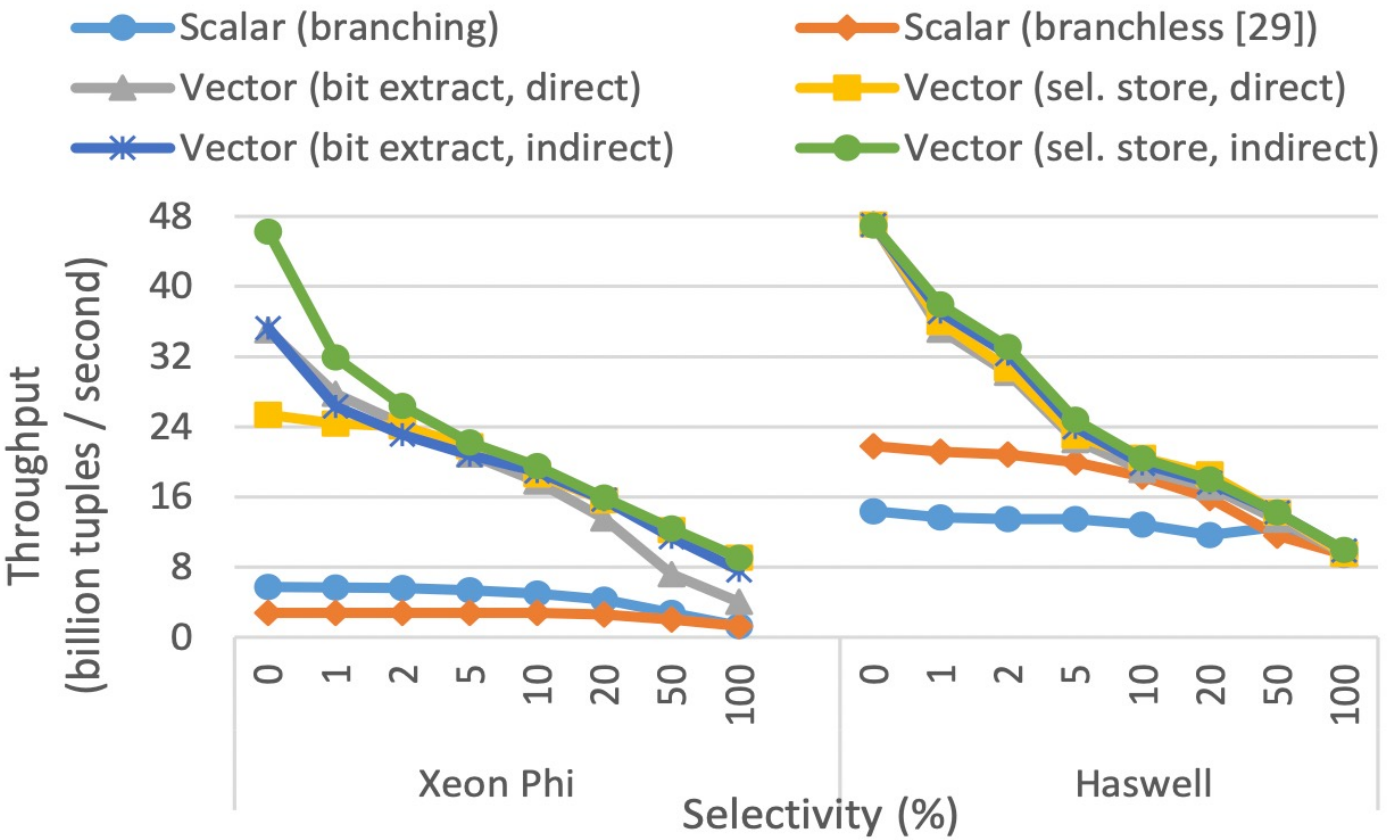


Figure 5: Selection scan (32-bit key & payload)

Hash Tables

Fig 6 shows Linear probing and double hashing.

Fig 7 shows probing throughput of Cuckoo hashing.

Fig 8 shows 1:1 interleaved build and probe of shared-nothing tables

Fig 9 shows 1:10 interleaved build and probe of shared-nothing tables.

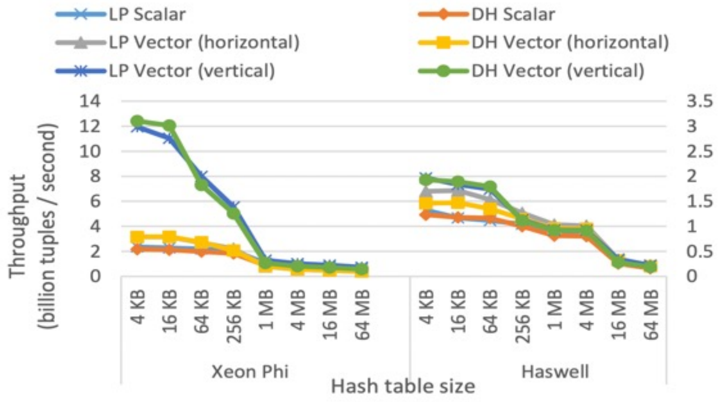


Figure 6: Probe linear probing & double hashing tables (shared, 32-bit key → 32-bit probed payload)

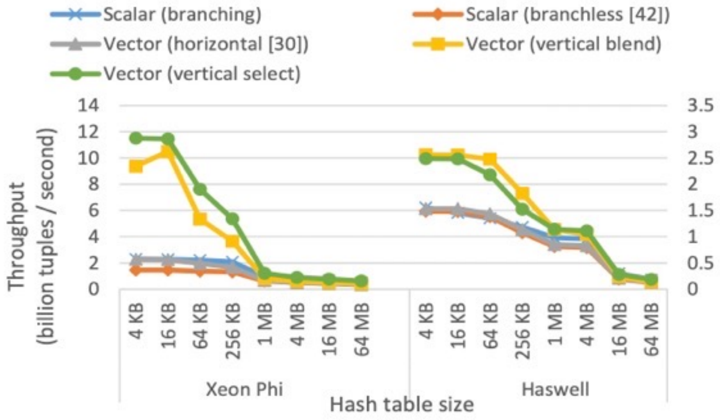


Figure 7: Probe cuckoo hashing table (2 functions, shared, 32-bit key → 32-bit probed payload)

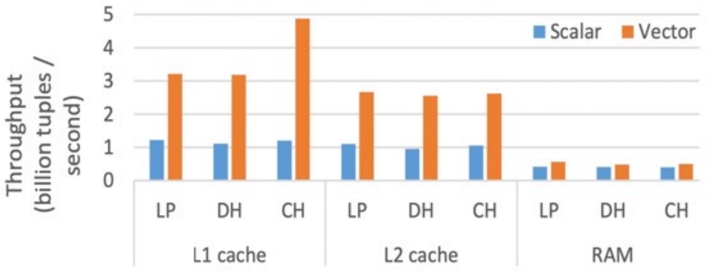


Figure 8: Build & probe linear probing, double hashing, & cuckoo hashing on Xeon Phi (1:1 build–probe, shared-nothing, 2X 32-bit key & payload)

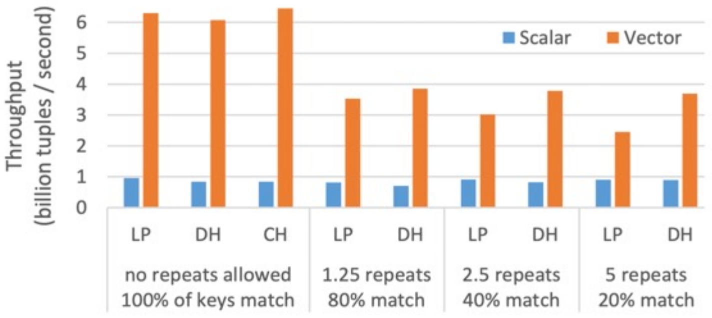


Figure 9: Build & probe linear probing, double hashing, & cuckoo hashing on Xeon Phi (1:10 build–probe, L1, shared-nothing, 2X 32-bit key & payload)

Bloom Filters

Fig 10 shows bloom filter probing throughput with selective loads and stores.

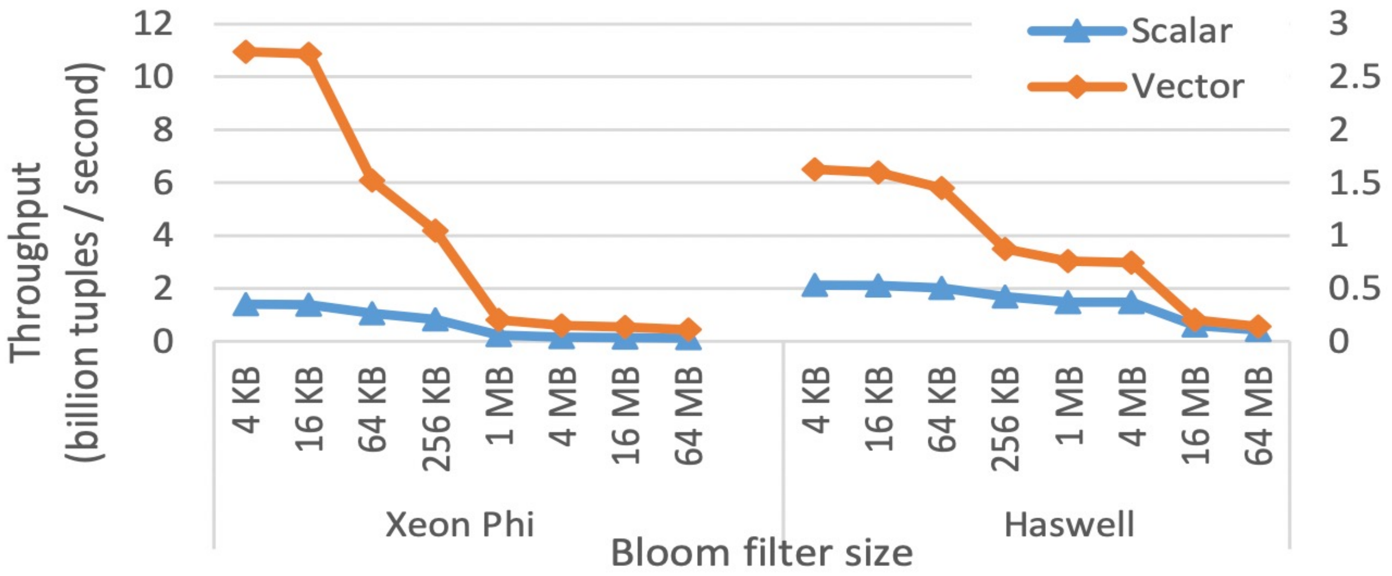


Figure 10: Bloom filter probing (5 functions, shared, 10 bits / item, 5% selectivity, 32-bit key & payload)

Partitioning

- Figure 11 shows radix and hash histogram generation on Xeon Phi.
- Figure 12 shows the performance of computing the range partition function.
- Figure 13 measures shuffling on Xeon Phi using inputs larger than the cache.

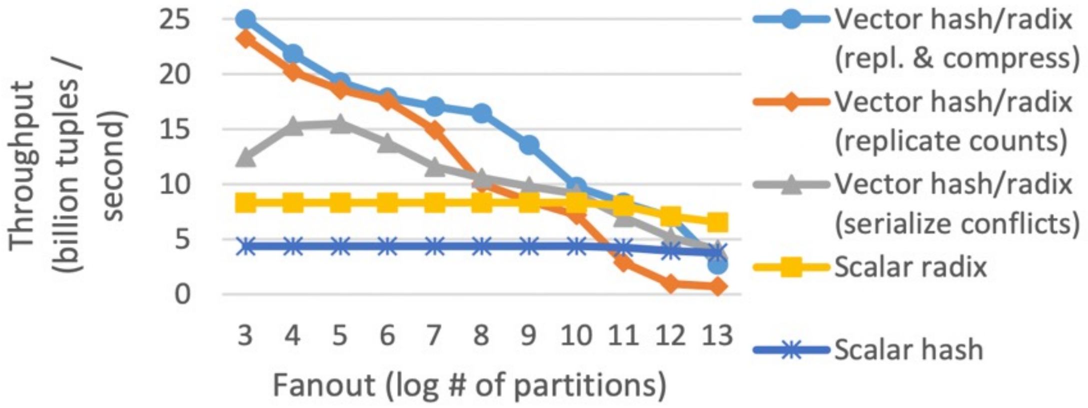


Figure 11: Radix & hash histogram on Xeon Phi

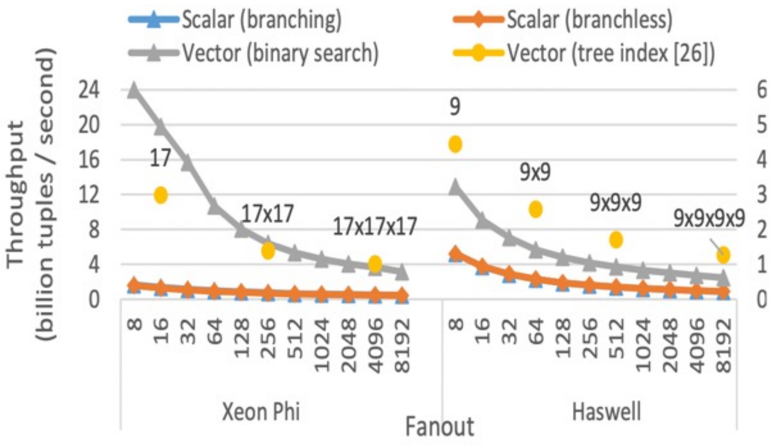


Figure 12: Range function on Xeon Phi (32-bit key)

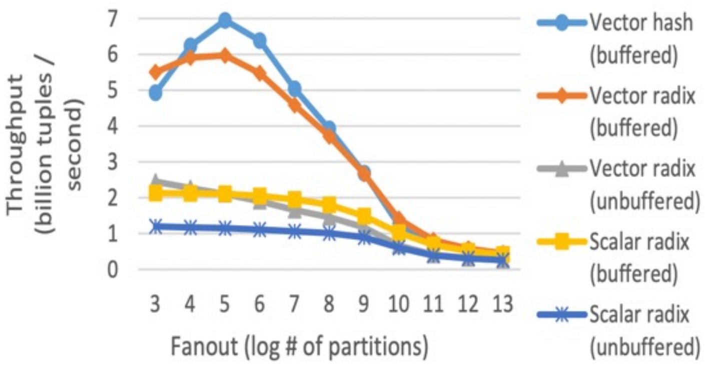


Figure 13: Radix shuffling on Xeon Phi (shared-nothing, out-of-cache, 32-bit key & payload)

Sorting and Hash Join

Figure 14 shows the performance of LSB radixsort on Xeon Phi.

Figure 15 shows the performance of the three hash join variants as described in Section 9, on Xeon Phi.

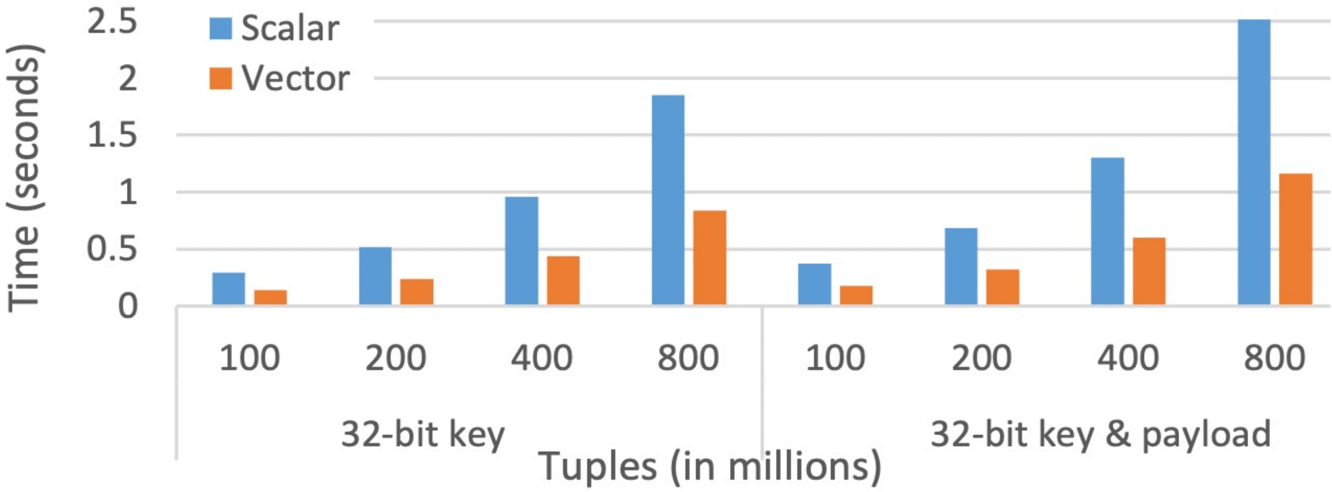


Figure 14: Radixsort on Xeon Phi (LSB)

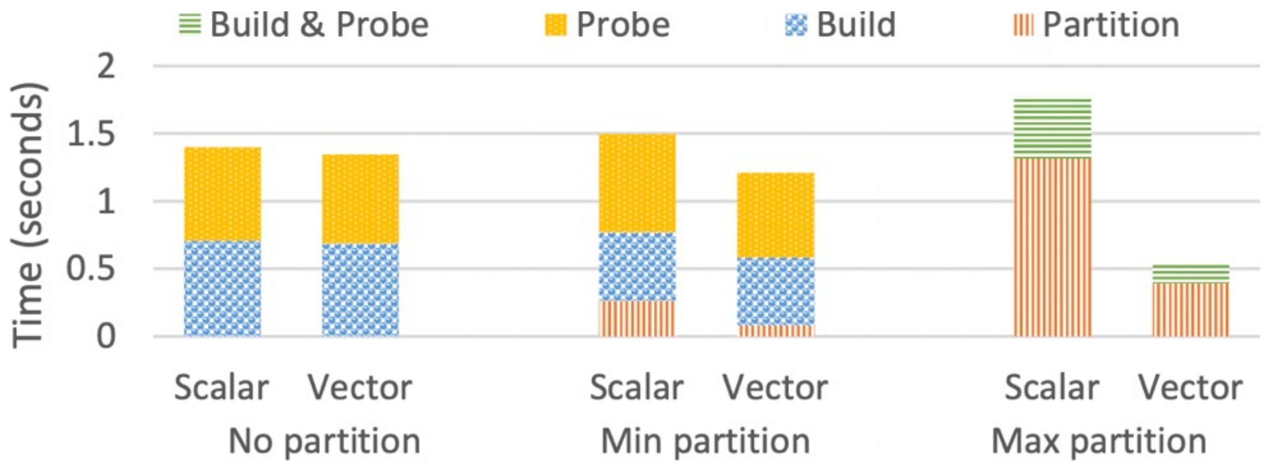


Figure 15: Multiple hash join variants on Xeon Phi ($2 \cdot 10^8 \bowtie 2 \cdot 10^8$ 32-bit key & payload)

Sorting and Hash Join

Figure 16 shows the thread scalability of radix sort and partitioned hash join on Xeon Phi.

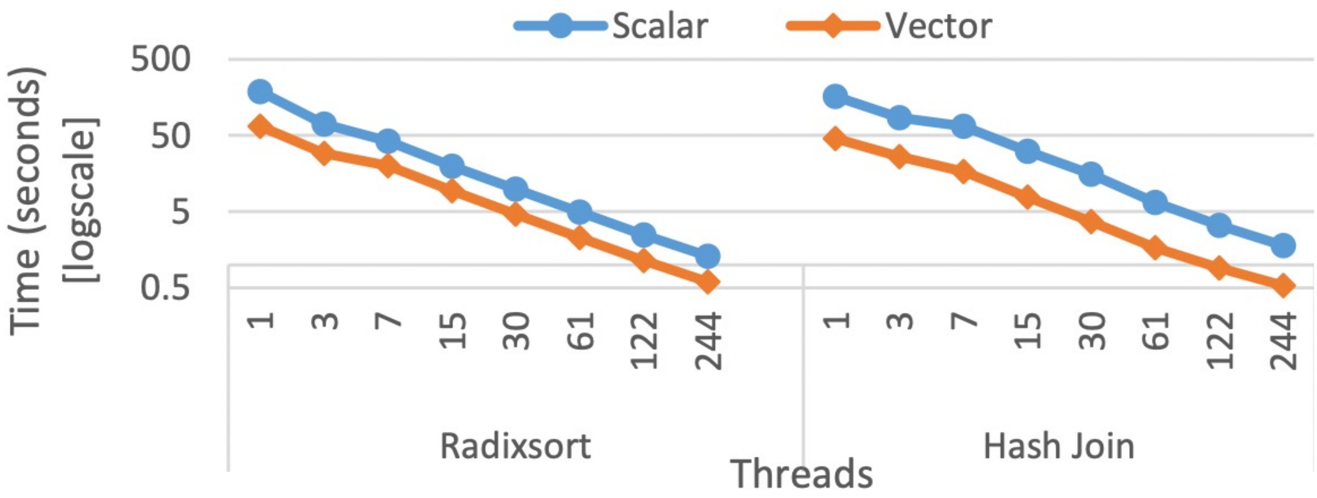


Figure 16: Radixsort & hash join scalability ($4 \cdot 10^8$ & $2 \cdot 10^8 \bowtie 2 \cdot 10^8$ 32-bit key & payload, log/log scale)

Sorting and Hash Join

We now compare Xeon Phi to 4 Sandy Bridge (SB) CPUs in order to get comparable performance, using radix sort and hash join.

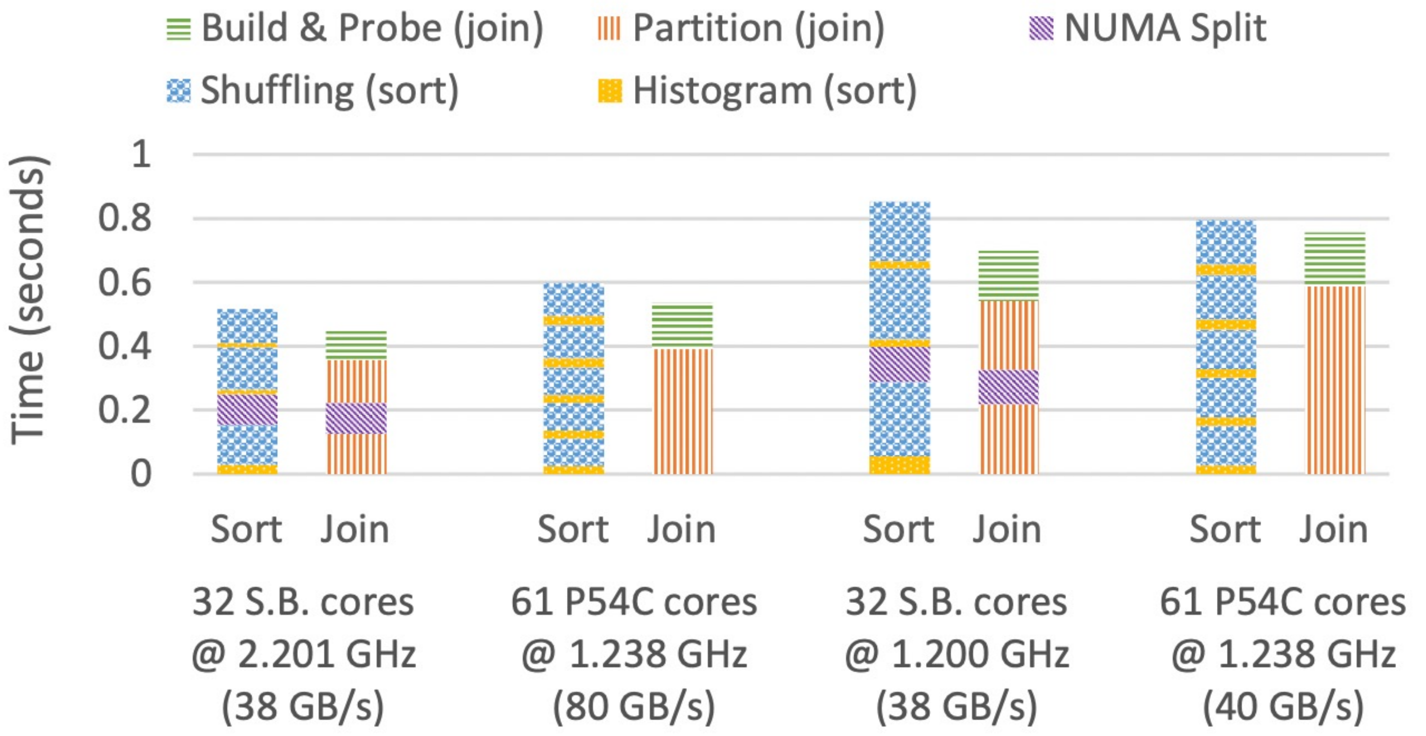


Figure 17: Radixsort & hash join on Xeon Phi 7120P versus 4 Xeon E5 4620 CPUs (sort $4 \cdot 10^8$ tuples, join $2 \cdot 10^8 \bowtie 2 \cdot 10^8$ tuples, 32-bit key & payload per table)

Sorting and Hash Join

Figure 18 measures radix sort with 32-bit keys by varying the number and width of payload columns.

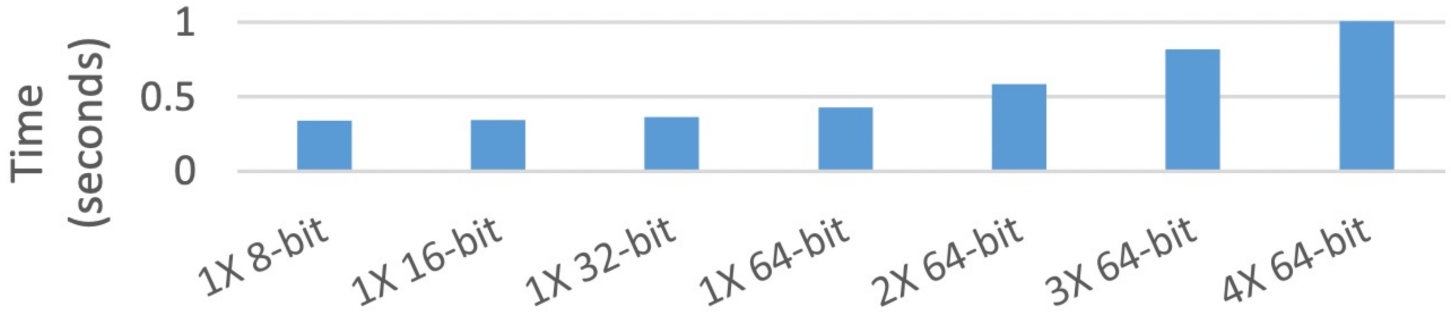


Figure 18: Radixsort with varying payloads on Xeon Phi ($2 \cdot 10^8$ tuples, 32-bit key)

Sorting and Hash Join

- Figure 19 shows partitioned hash join with 32-bit keys and multiple 64-bit payload columns.

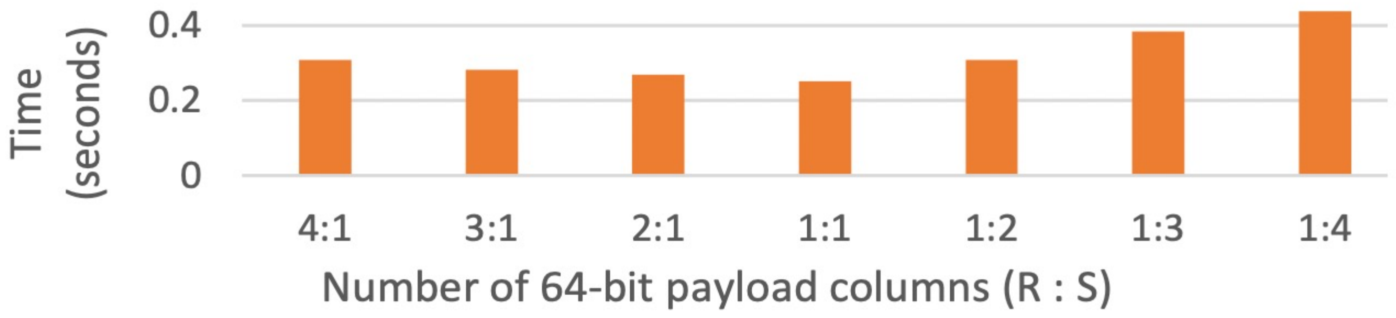


Figure 19: Hash join with varying payload columns on Xeon Phi ($10^7 \bowtie 10^8$ tuples, 32-bit keys)