Rethinking SIMD Vectorization for In-Memory Databases

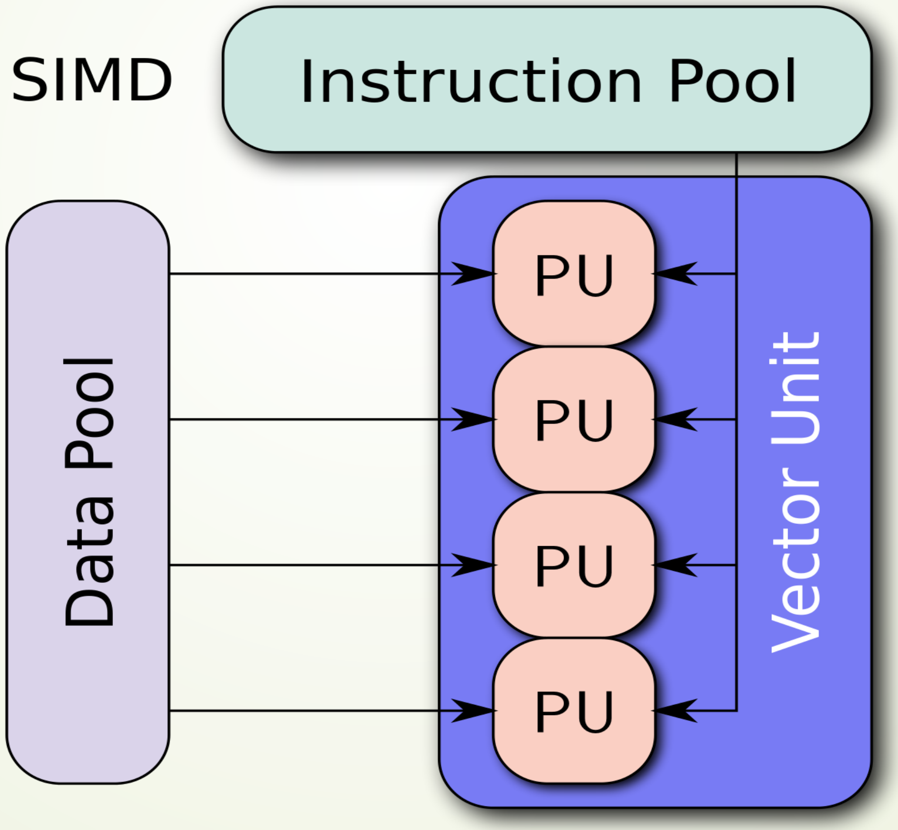
Sri Harshal Parimi

Motivation

- Need for fast analytical query execution in systems where the database is mostly resident in main memory.
- Architectures with SIMD capabilities, like (Many Integrated cores)MIC, use a large number of *low-powered cores* with advanced instruction sets and larger registers.

SIMD(Single Instruction, Multiple Data)

- Multiple processing elements that perform the same operation on multiple data points simultaneously.



Vectorization

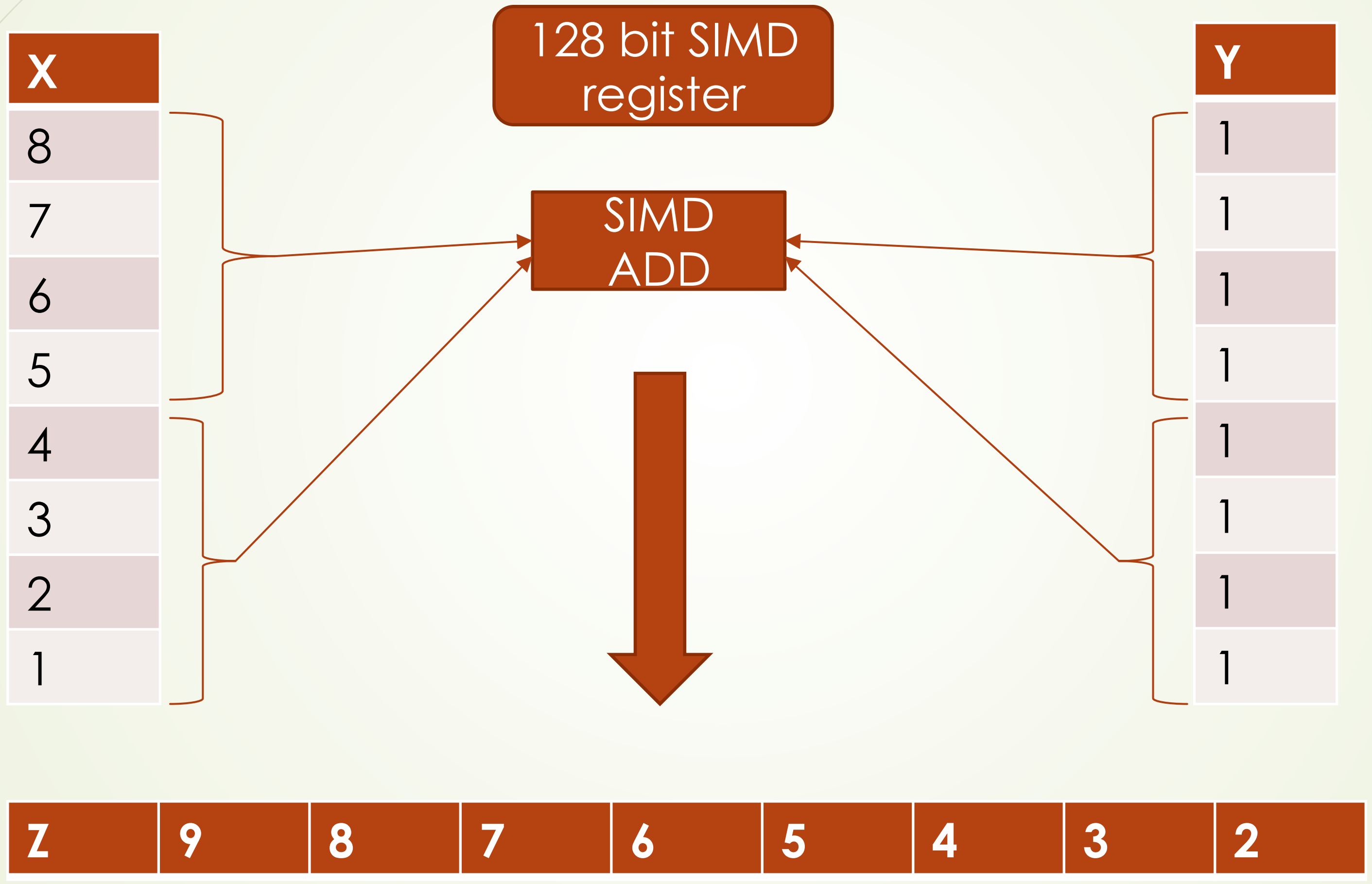
- Program that performs operations on a vector(1D- array).

$X+Y=Z$

$(x_1, x_2, \ldots, x_n) + (y_1, y_2, \ldots, y_n) = (x_1+y_1, x_2+y_2, \ldots, x_n+y_n)$

```
for(i = 0; i<n; i++){
        Z[i] = X[i] + Y[i];
}
```

Vectorization(Example)



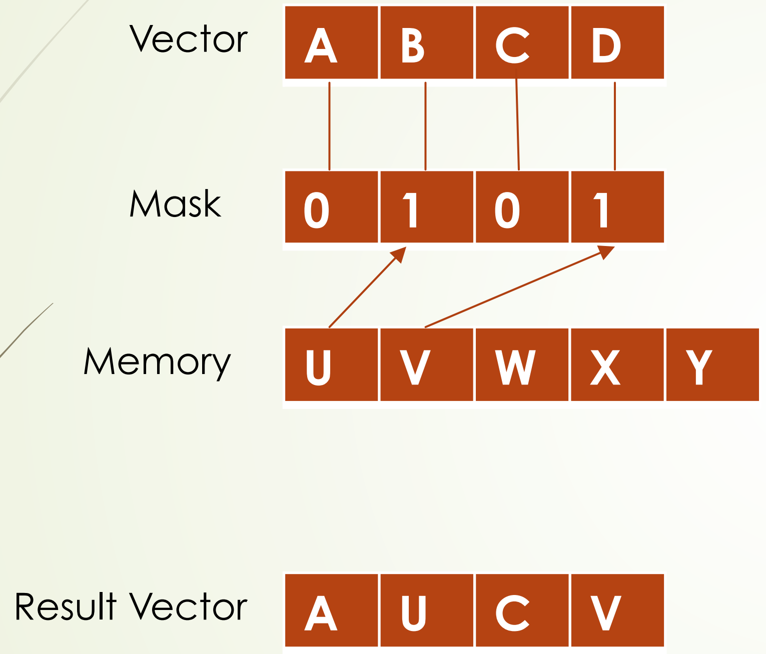
Advantages of Vectorization

- Full vectorization
  - From O(f(n)) scalar to O(f(n)/W) vector operations where W is the length of the vector.
  - Reuse ***fundamental operations*** across multiple vectorizations.
- Vectorize basic database operators:
  - Selection scans
  - Hash tables
  - Partitioning

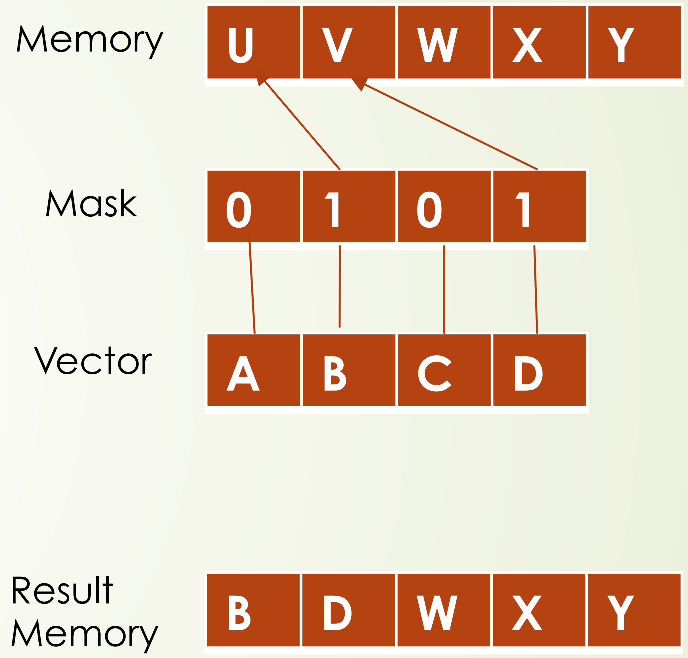
Fundamental Operations

- Selective Load
- Selective Store
- Selective Gather
- Selective Scatter

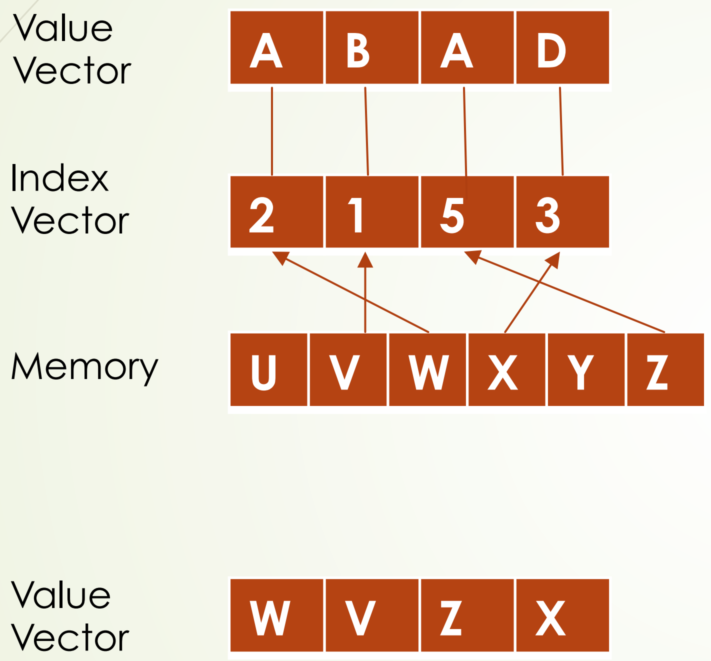
Selective Load



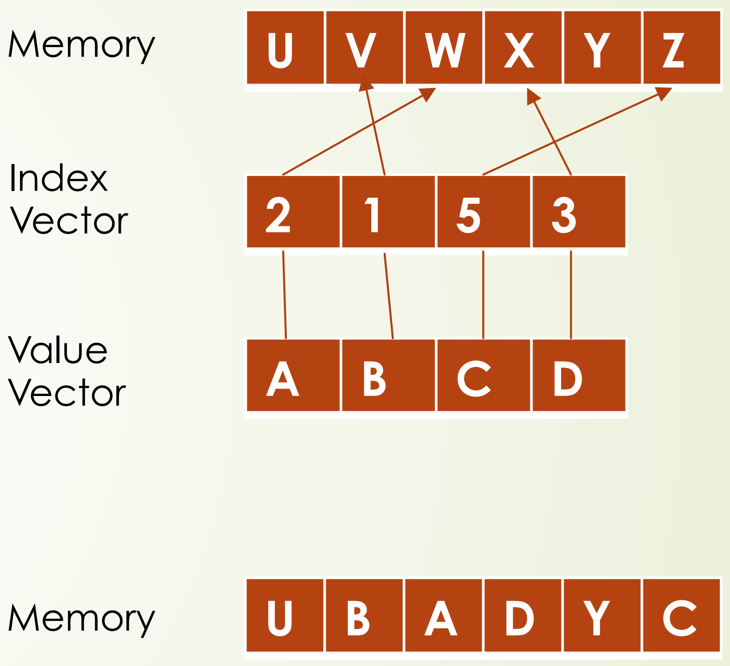
Selective Store



Selective Gather



Selective Scatter



Selection Scans

Scalar(Branching):

- I = 0
- For t in table:
  - If((key>= “O” && key<=“U”)):
    - Copy(t, output[i]);
    - I = I + 1;

Scalar(Branchless):

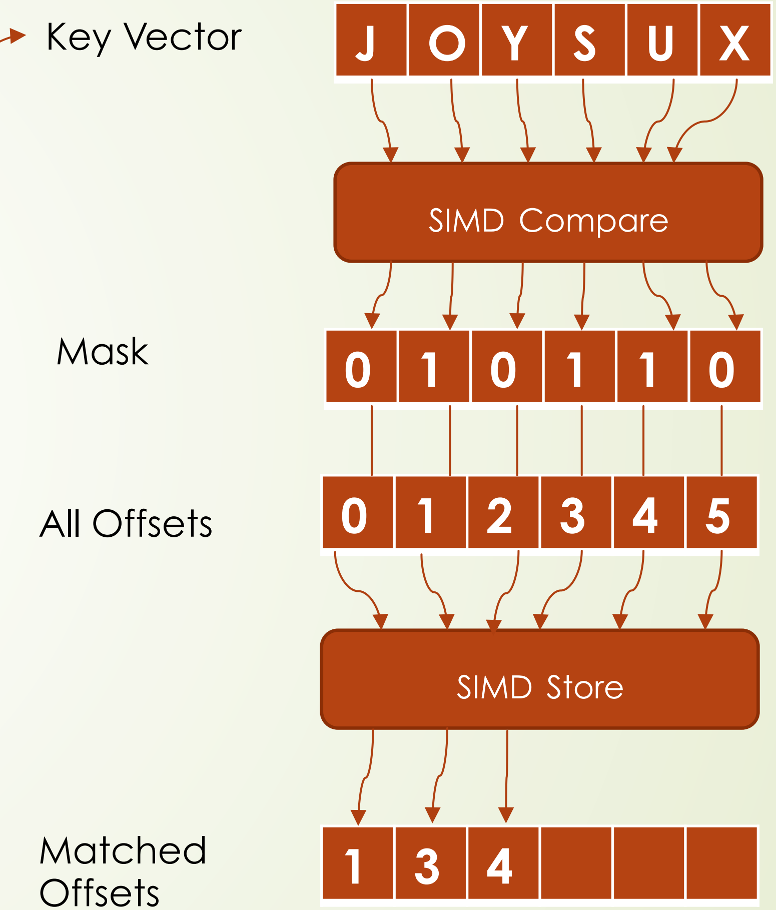
- I = 0
- For t in table:
  - Key = t.key
  - M = (key>=“O”?1:0)&&(key<=“U”?1:0);
  - I = I + M;

SELECT * FROM table WHERE key >=“O” AND key<=“U”

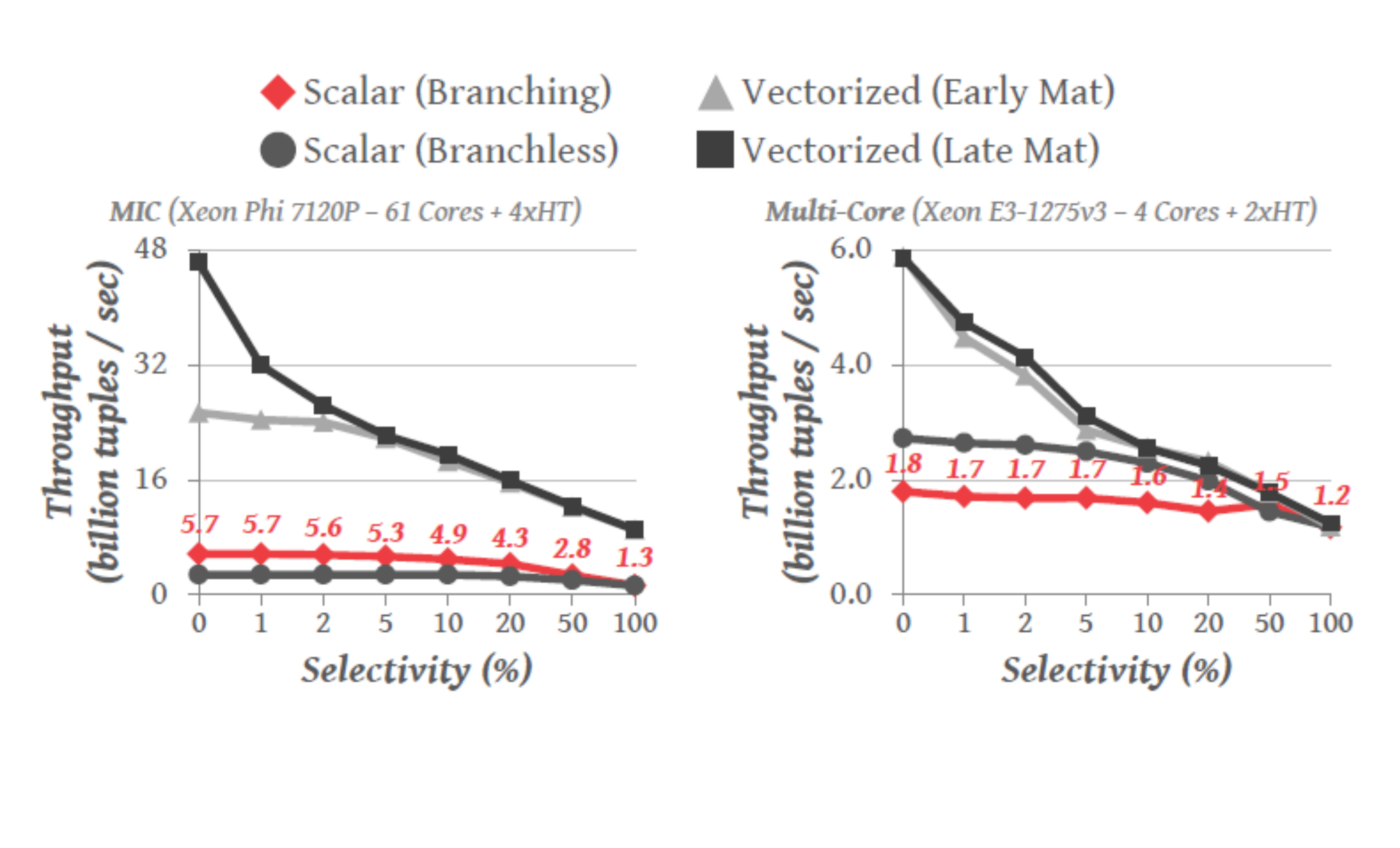
Selection Scans(Vectorized)

- I = 0
- For V_t in table:
  - simdLoad(V_t.key, V_k)
  - $V_m = (V_k >= \text{“O”} ? 1:0) \&\& (V_k <= \text{“U”} ? 1:0)$
  - If(V_m != false):
    - simdStore(V_t, V_m, output[i])
    - I = I + |V_m!= false|

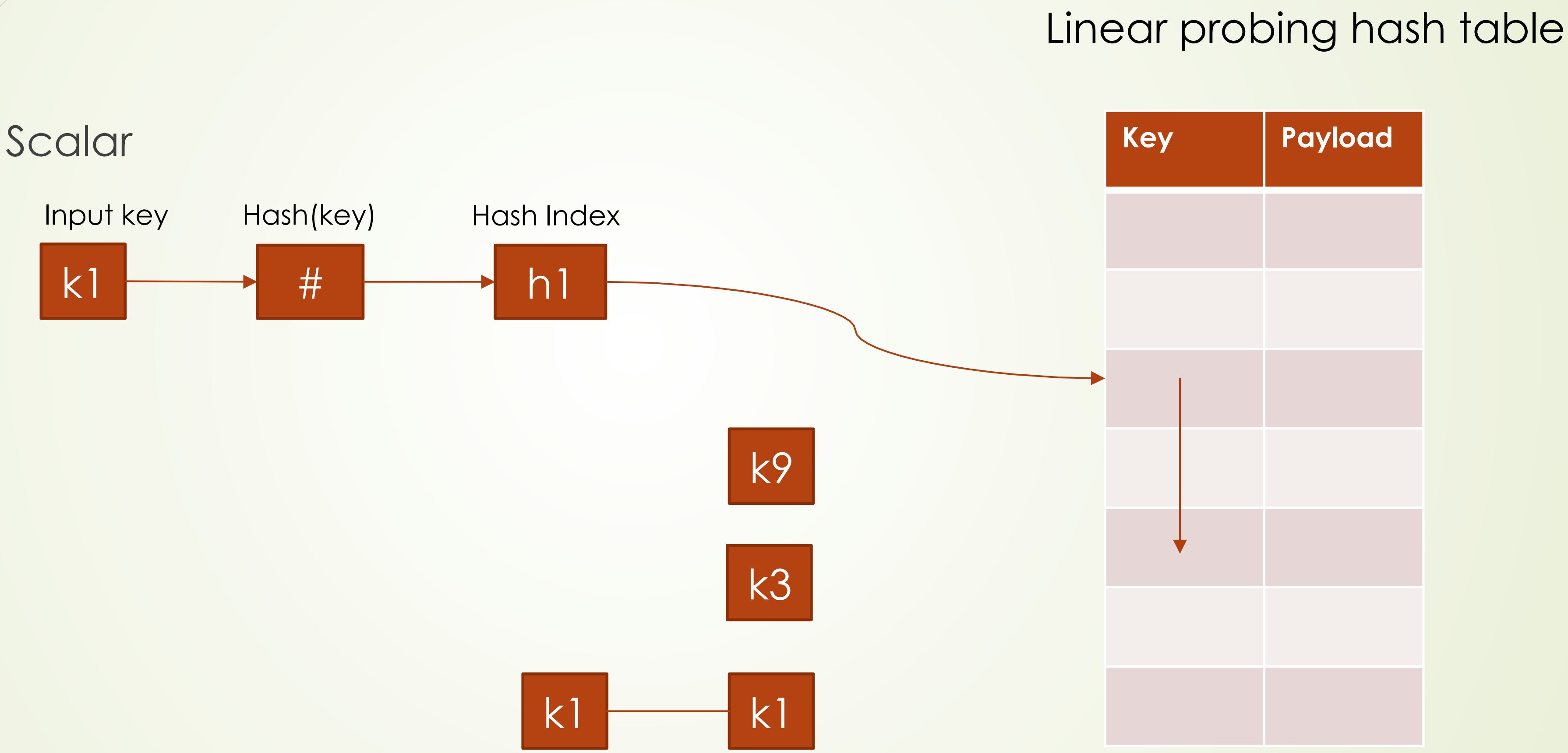
ID	KEY
1	J
2	O
3	Y
4	S
5	U
6	X



Performance Comparison: Selection Scans

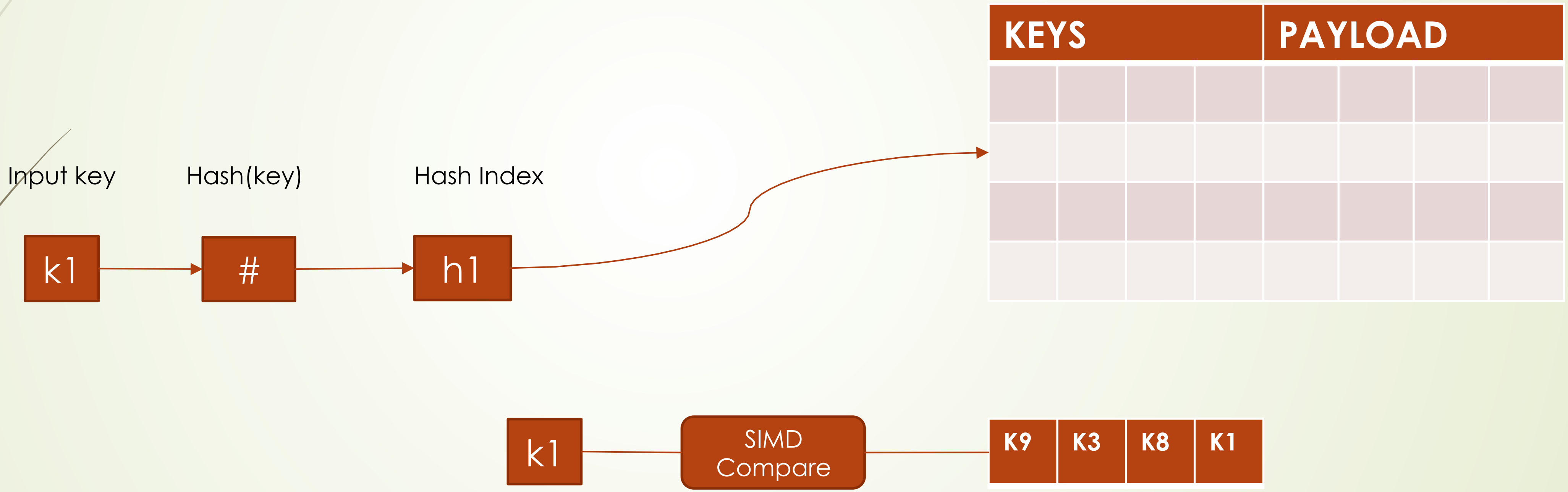


Hash Tables – Probing (Scalar)

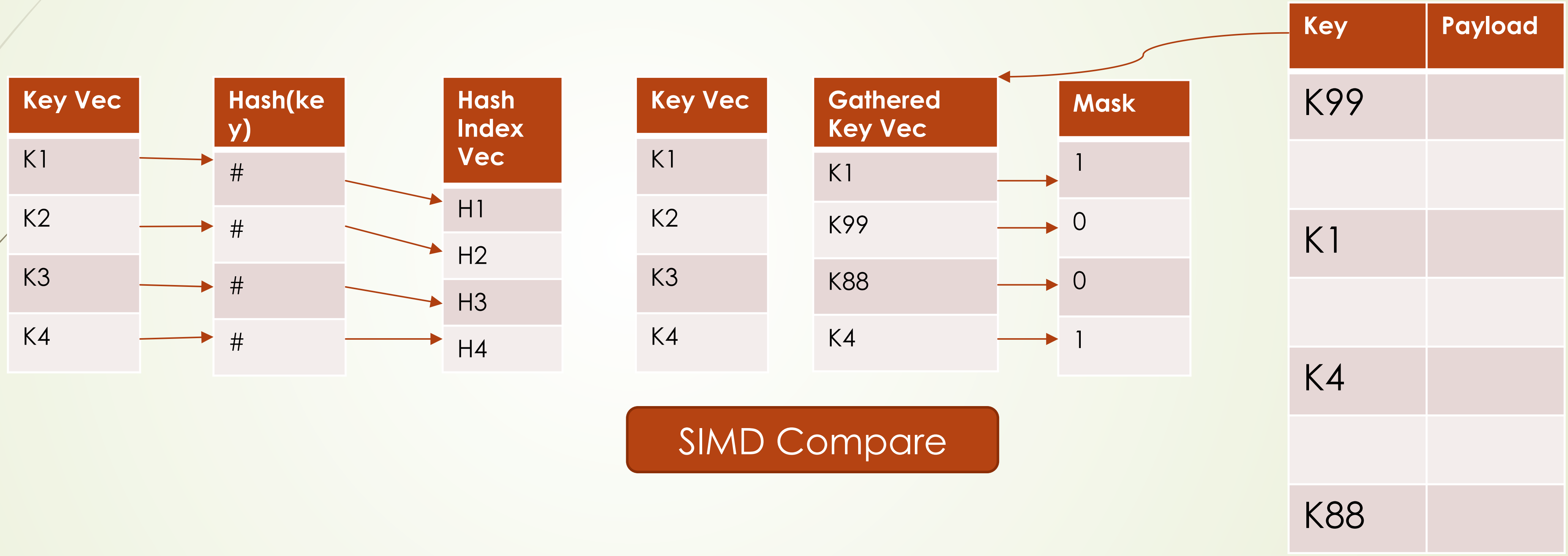


Hash Tables – Probing (Horizontal Vectorization)

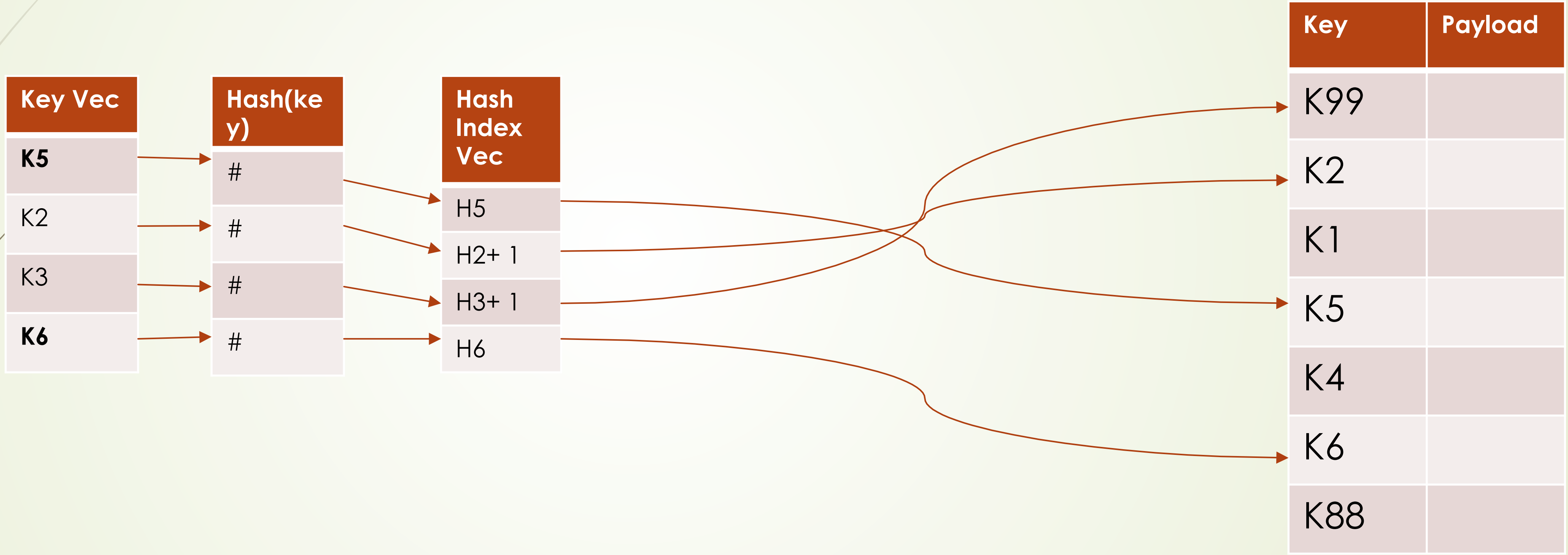
Linear probing bucketized hash table



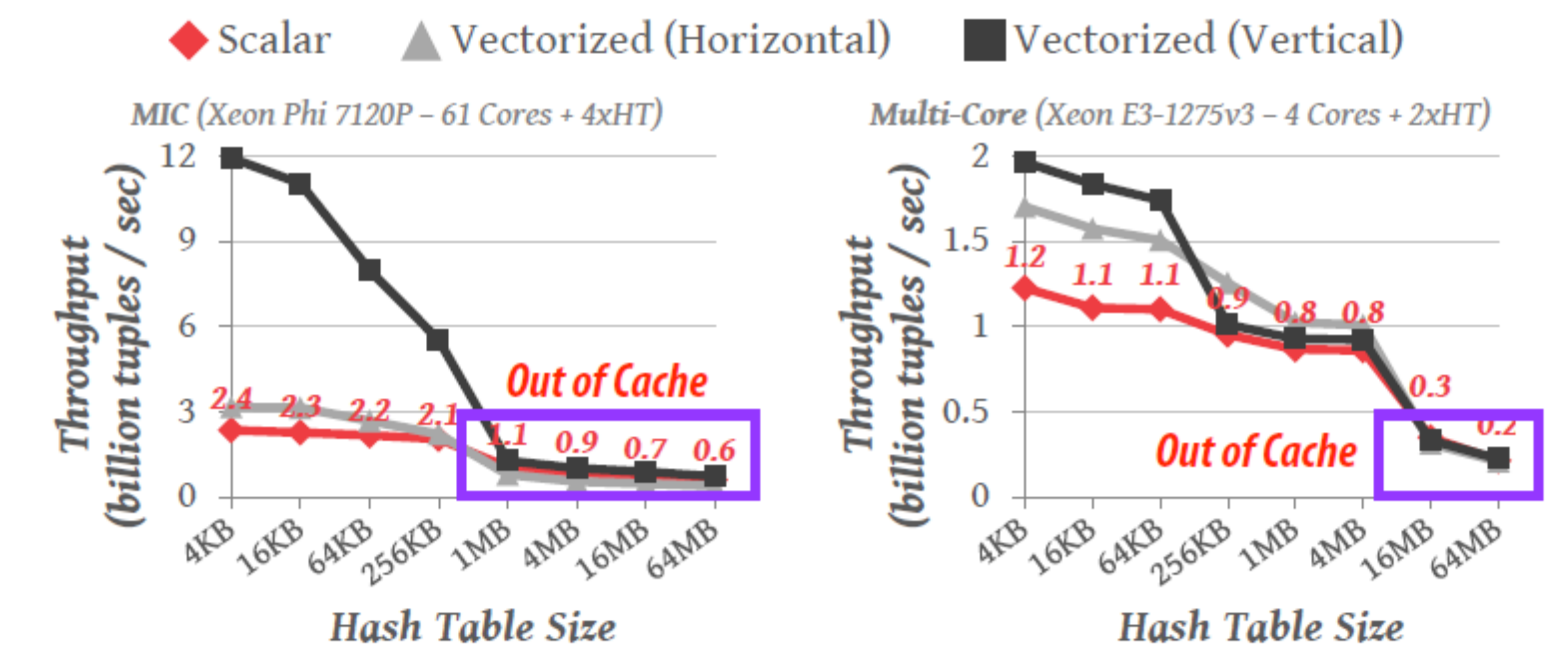
Hash Tables – Probing (Vertical Vectorization)



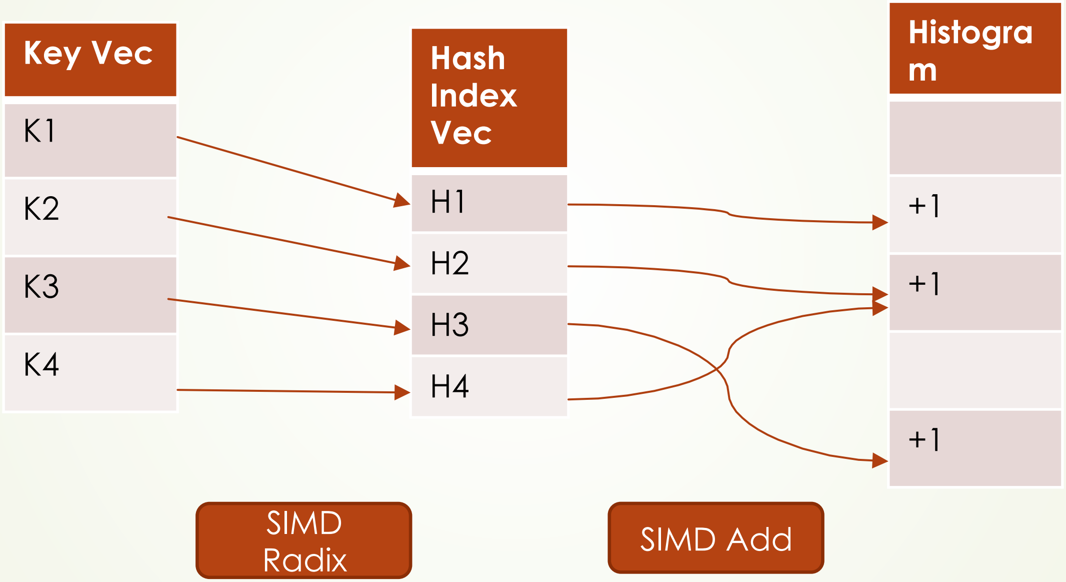
Hash Tables – Probing (Vertical Vectorization Continued)



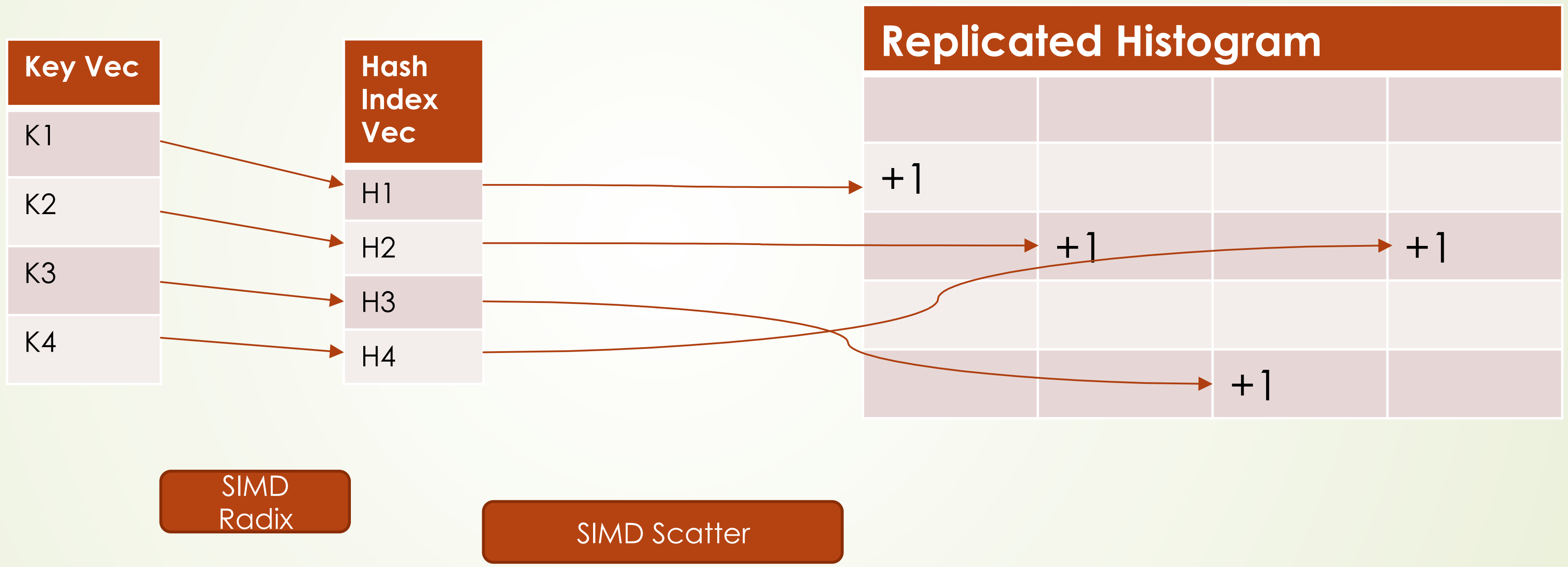
Performance Comparison: Hash Tables



Partitioning - Histogram



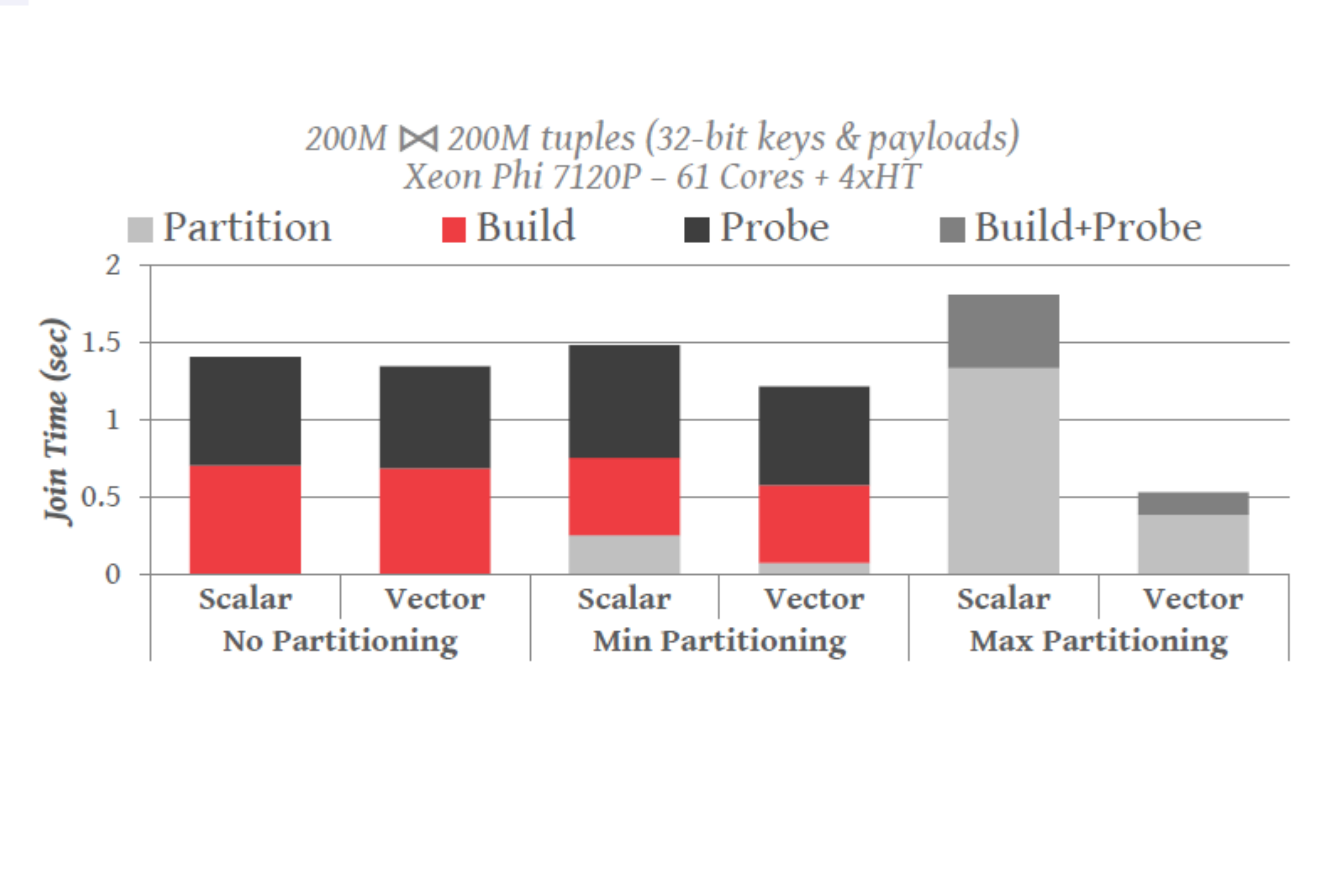
Partitioning – Histogram(Continued)



Joins

- No partitioning
  - Build one shared hash table using atomics
  - Partially vectorized
- Min partitioning
  - Partition building table
  - Build hash table per thread
  - Fully vectorized
- Max partitioning
  - Partition both tables repeatedly
  - Build and probe **cache-resident** hash tables
  - Fully vectorized

Joins


200M ⋈ 200M tuples (32-bit keys & payloads)
Xeon Phi 7120P – 61 Cores + 4xHT
Partition
Build
Probe
Build+Probe
Join Time (sec)
2
1.5
1
0.5
0
Scalar
Vector
No Partitioning
Scalar
Vector
Min Partitioning
Scalar
Vector
Max Partitioning

Main Takeaways

- Vectorization is essential for OLAP queries
- Impact on hardware design
  - Improved power efficiency for analytical databases
- Impact on software design
  - Vectorization favors cache-conscious algorithms
    - Partitioned hash join >> non-partitioned hash join, if vectorized
  - Vectorization is independent of other optimizations
    - Both buffered and unbuffered partitioning benefit from vectorization speedup

Comparisons with Trill

- Trill uses a similar bit-mask technique for applying the filter clause during selections.
- While Trill deals with a query model for streaming data, this paper offers algorithms that can improve throughput of database operators which can also be extended to a streaming model by leveraging buffered data.
- Trill uses dynamic HLL code-generation to operate over columnar data. SIMD provides vectorization to handle data-points simultaneously and has a diverse instruction set(supported by H/W) to perform constant operations on vectors.

Questions?