CS 6530: Advanced Database Systems Fall 2022

Lecture 19
Vectorization

Prashant Pandey

prashant.pandey@utah.edu

Acknowledgement: Slides taken from Prof. Andy Pavlo, CMU

SCHOOL OF COMPUTING
UNIVERSITY OF UTAH

VECTORIZATION

- The process of converting an algorithm's scalar implementation that processes a single pair of operands at a time, to a vector implementation that processes one operation on multiple pairs of operands at once.

WHY THIS MATTERS

- Say we can parallelize our algorithm over 32 cores.
- Each core has a 4-wide SIMD registers.

- **Potential Speed-up: 32x × 4x = 128x**

SINGLE INSTRUCTION, MULTIPLE DATA

- A class of CPU instructions that allow the processor to perform the same operation on multiple data points simultaneously.

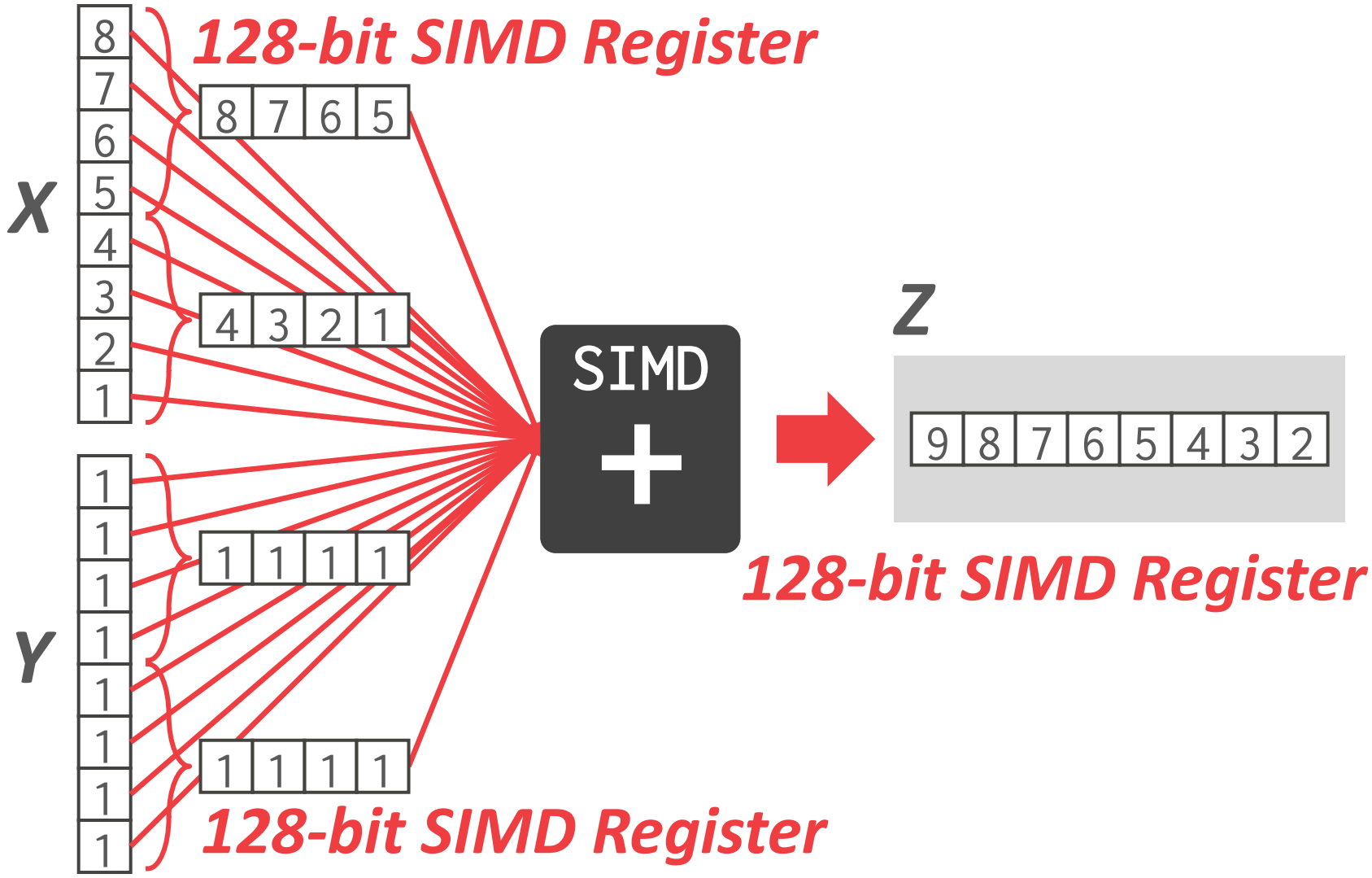
- All major ISAs have microarchitecture support SIMD operations.
  - **x86**: MMX, SSE, SSE2, SSE3, SSE4, AVX, AVX2, AVX512
  - **PowerPC**: Altivec
  - **ARM**: NEON, SVE

SIMD EXAMPLE

$$X + Y = Z$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

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



SIMD INSTRUCTIONS (1)

- **Data Movement**
  - Moving data in and out of vector registers
- **Arithmetic Operations**
  - Apply operation on multiple data items (e.g., 2 doubles, 4 floats, 16 bytes)
  - Example: `ADD`, `SUB`, `MUL`, `DIV`, `SQRT`, `MAX`, `MIN`
- **Logical Instructions**
  - Logical operations on multiple data items
  - Example: `AND`, `OR`, `XOR`, `ANDN`, `ANDPS`, `ANDNPS`

SIMD INSTRUCTIONS (2)

- **Comparison Instructions**
  - Comparing multiple data items (==,<,<=,>,>=,!=)
- **Shuffle instructions**
  - Move data in between SIMD registers
- **Miscellaneous**
  - Conversion: Transform data between x86 and SIMD registers.
  - Cache Control: Move data directly from SIMD registers to memory (bypassing CPU cache).

INTEL SIMD EXTENSIONS

		Width	*Integers*	*Single-P*	*Double-P*
1997	MMX	64 bits	✓		
1999	SSE	128 bits	✓	✓(×4)	
2001	SSE2	128 bits	✓	✓	✓(×2)
2004	SSE3	128 bits	✓	✓	✓
2006	SSSE 3	128 bits	✓	✓	✓
2006	SSE 4.1	128 bits	✓	✓	✓
2008	SSE 4.2	128 bits	✓	✓	✓
2011	AVX	256 bits	✓	✓(×8)	✓(×4)
2013	AVX2	256 bits	✓	✓	✓
2017	**AVX-512**	**512 bits**	✓	✓**(×16)**	✓**(×8)**

Source: James Reinders

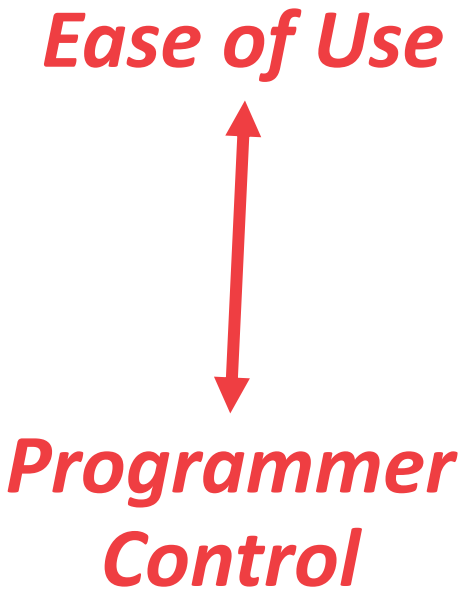
SIMD TRADE-OFFS

- **Advantages:**
  - Significant performance gains and resource utilization if an algorithm can be vectorized.

- **Disadvantages:**
  - Implementing an algorithm using SIMD is still mostly a manual process.
  - SIMD may have restrictions on data alignment.
  - Gathering data into SIMD registers and scattering it to the correct locations is tricky and/or inefficient.

VECTORIZATION

- **Choice #1: Automatic Vectorization**
- **Choice #2: Compiler Hints**
- **Choice #3: Explicit Vectorization**



Source: James Reinders

AUTOMATIC VECTORIZATION

- The compiler can identify when instructions inside of a loop can be rewritten as a vectorized operation.
- Works for simple loops only and is rare in database operators. Requires hardware support for SIMD instructions.

AUTOMATIC VECTORIZATION

```
void add(int *X,
         int *Y,
         int *Z) {
  for (int i=0; i<MAX; i++) {
    Z[i] = X[i] + Y[i];
  }
}
```

*Z=*X+1

These might point to the same address!

- This loop is not legal to automatically vectorize.
- The code is written such that the addition is described sequentially.

COMPILER HINTS

- Provide the compiler with additional information about the code to let it know that is safe to vectorize.

- Two approaches:
  - Give explicit information about memory locations.
  - Tell the compiler to ignore vector dependencies.

COMPILER HINTS

```
void add(int *restrict X,
         int *restrict Y,
         int *restrict Z) {
  for (int i=0; i<MAX; i++) {
    Z[i] = X[i] + Y[i];
  }
}
```

- The `restrict` keyword in C++ tells the compiler that the arrays are distinct locations in memory.

COMPILER HINTS

```
void add(int *X,
         int *Y,
         int *Z) {
#pragma ivdep
  for (int i=0; i<MAX; i++) {
    Z[i] = X[i] + Y[i];
  }
}
```

- This pragma tells the compiler to ignore loop dependencies for the vectors.
- It’s up to you make sure that this is correct.

EXPLICIT VECTORIZATION

- Use CPU intrinsics to manually marshal data between SIMD registers and execute vectorized instructions.

- Potentially not portable.

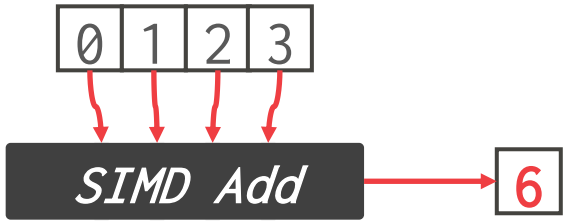
EXPLICIT VECTORIZATION

```
void add(int *X,
         int *Y,
         int *Z) {
  __mm128i *vecX = (__m128i*)X;
  __mm128i *vecY = (__m128i*)Y;
  __mm128i *vecZ = (__m128i*)Z;
  for (int i=0; i<MAX/4; i++) {
    _mm_store_si128(vecZ++,
      _mm_add_epi32(*vecX++,
                    *vecY++));
  }
}
```

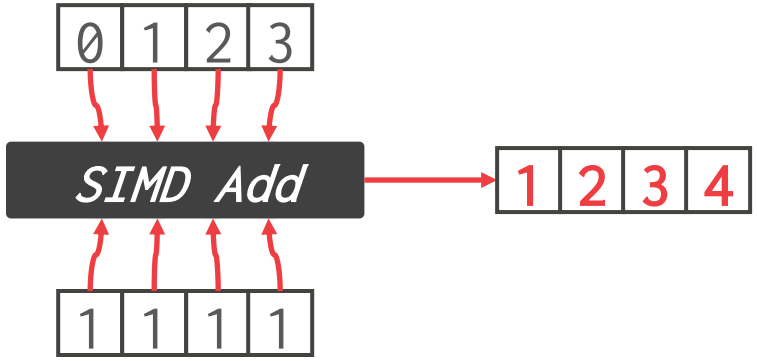
- Store the vectors in 128-bit SIMD registers.
- Then invoke the intrinsic to add together the vectors and write them to the output location.

VECTORIZATION DIRECTION

- **Approach #1: Horizontal**
  - Perform operation on all elements together within a single vector.



- **Approach #2: Vertical**
  - Perform operation in an elementwise manner on elements of each vector.



Source: Przemysław Karpiński

EXPLICIT VECTORIZATION

- **Linear Access Operators**
  - Predicate evaluation
  - Compression
- **Ad-hoc Vectorization**
  - Sorting
  - Merging
- **Composable Operations**
  - Multi-way trees
  - Bucketized hash tables

Source: Orestis Polychroniou

VECTORIZED DBMS ALGORITHMS

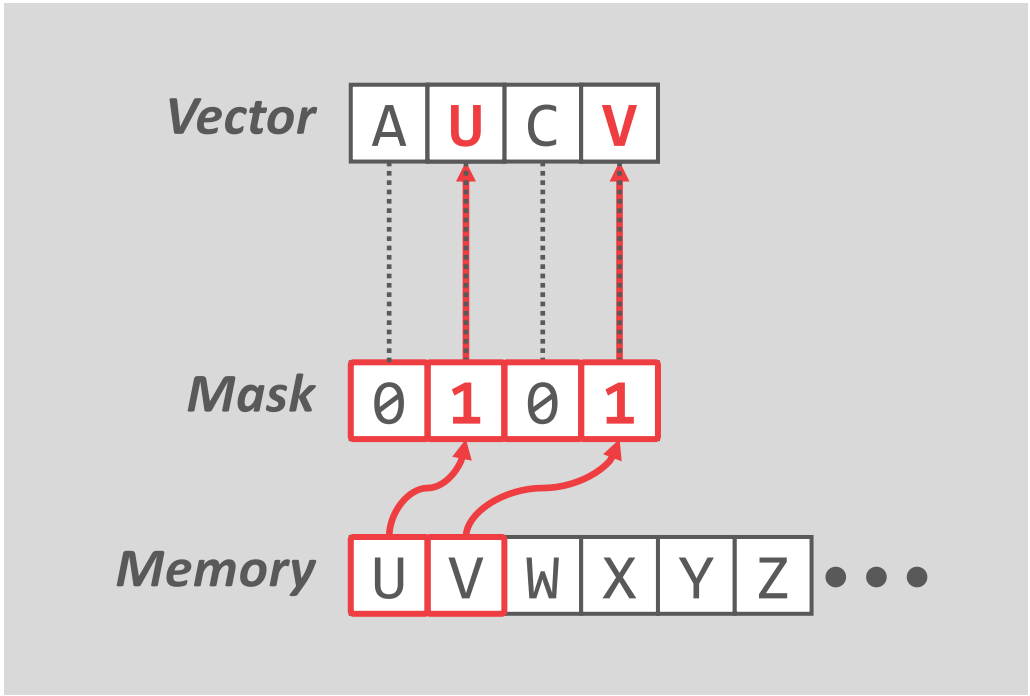
- Principles for efficient vectorization by using fundamental vector operations to construct more advanced functionality.
  - Favor ***vertical*** vectorization by processing different input data per lane.
  - Maximize lane utilization by executing unique data items per lane subset (i.e., no useless computations).

FUNDAMENTAL OPERATIONS

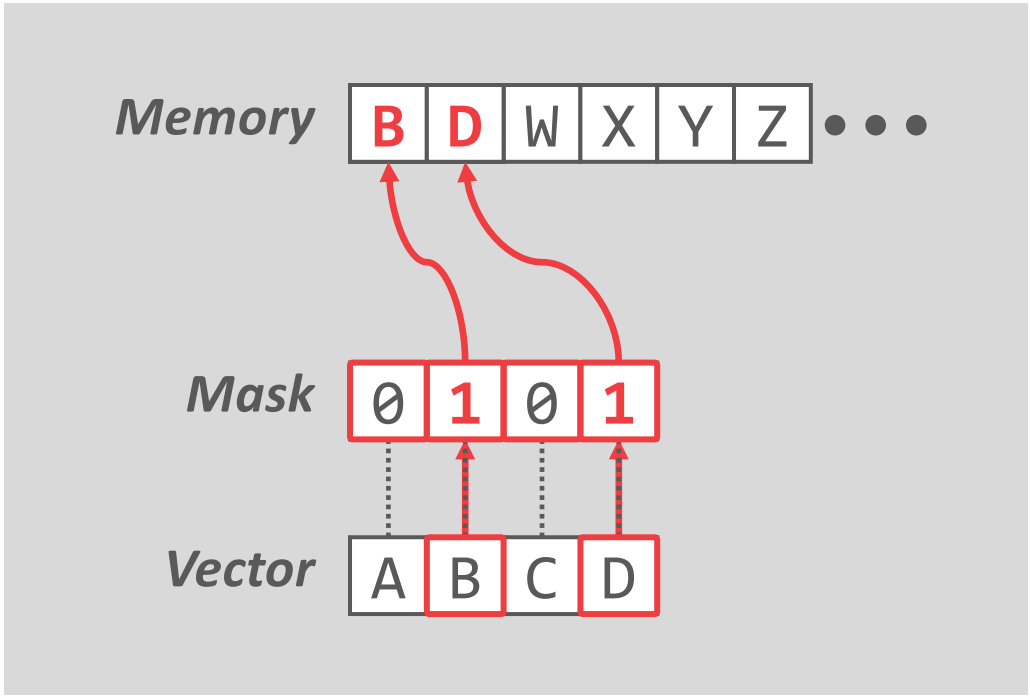
- Selective Load
- Selective Store
- Selective Gather
- Selective Scatter

FUNDAMENTAL VECTOR OPERATIONS

Selective Load

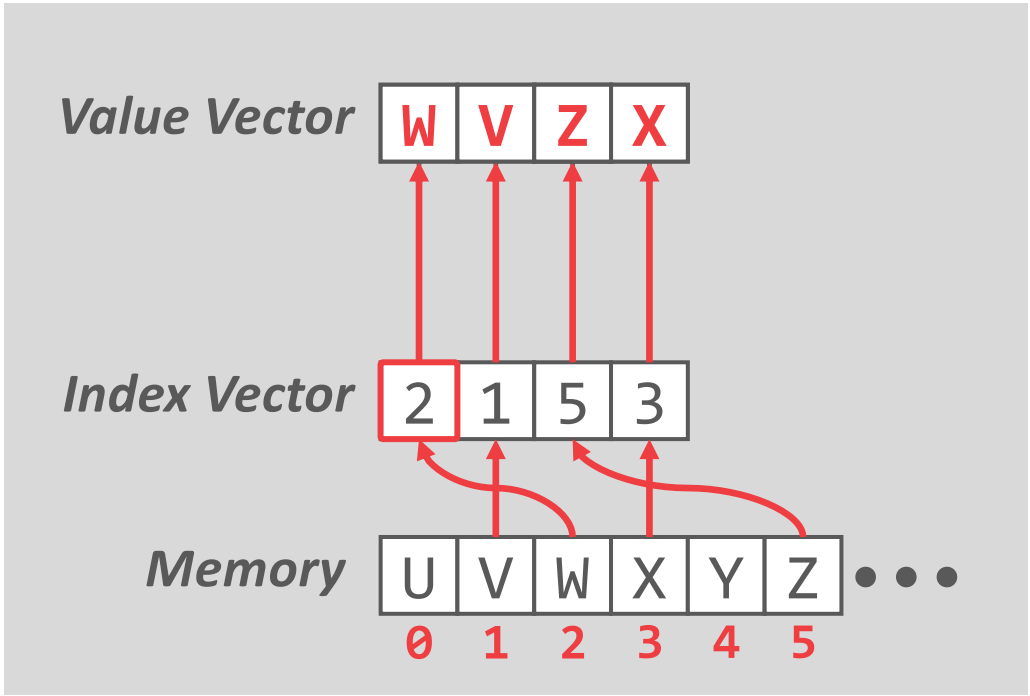


Selective Store

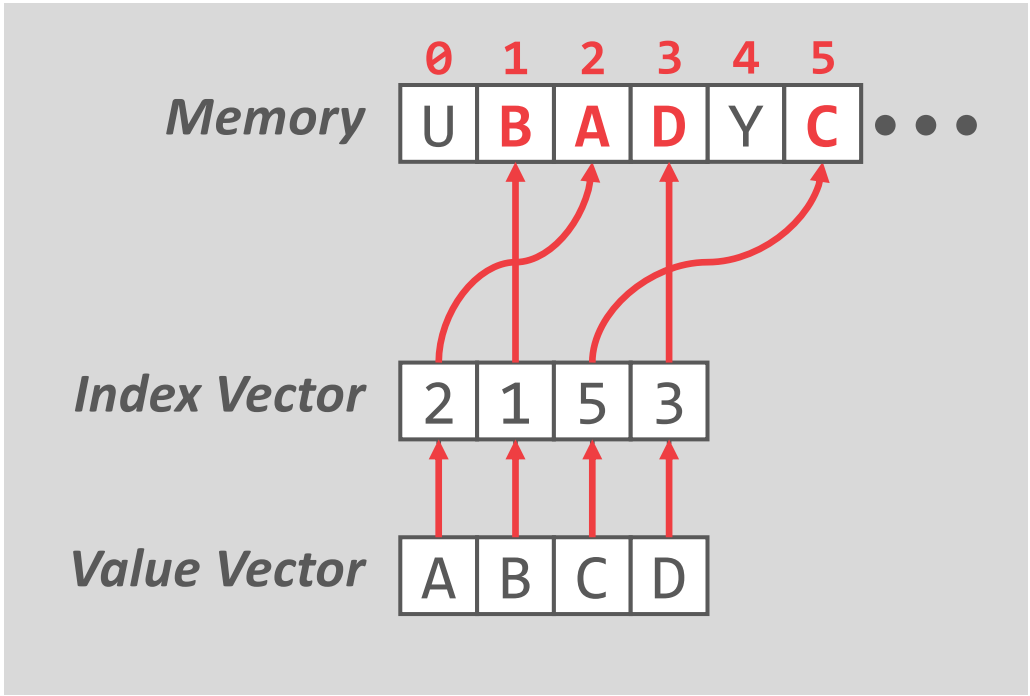


FUNDAMENTAL VECTOR OPERATIONS

Selective Gather



Selective Scatter



ISSUES

- Gathers and scatters are not really executed in parallel because the L1 cache only allows one or two distinct accesses per cycle.
- Gathers are only supported in newer CPUs.
- Selective loads and stores are also implemented in Xeon CPUs using vector permutations.

VECTORIZED OPERATORS

- Selection Scans
- Hash Tables
- Partitioning / Histograms

- Paper provides additional vectorized algorithms:
  - Joins, Sorting, Bloom filters.

SELECTION SCANS

Scalar (Branching)

```
i = 0
for t in table:
  key = t.key
  if (key≥low) && (key≤high):
    copy(t, output[i])
    i = i + 1
```

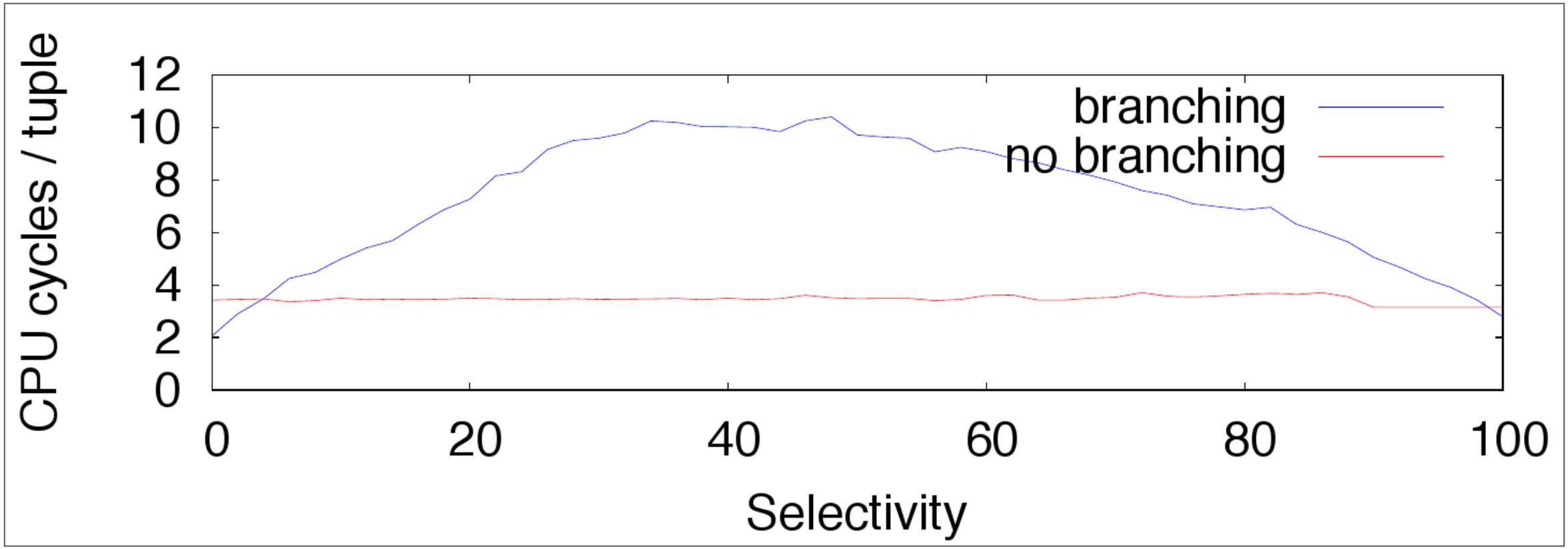
Scalar (Branchless)

```
i = 0
for t in table:
  copy(t, output[i])
  key = t.key
  m = (key≥low ? 1 : 0) &
      (key≤high ? 1 : 0)
  i = i + m
```

```
SELECT * FROM table
 WHERE key >= $low AND key <= $high
```

Source: Bogdan Raducanu

SELECTION SCANS



Source: Bogdan Raducanu

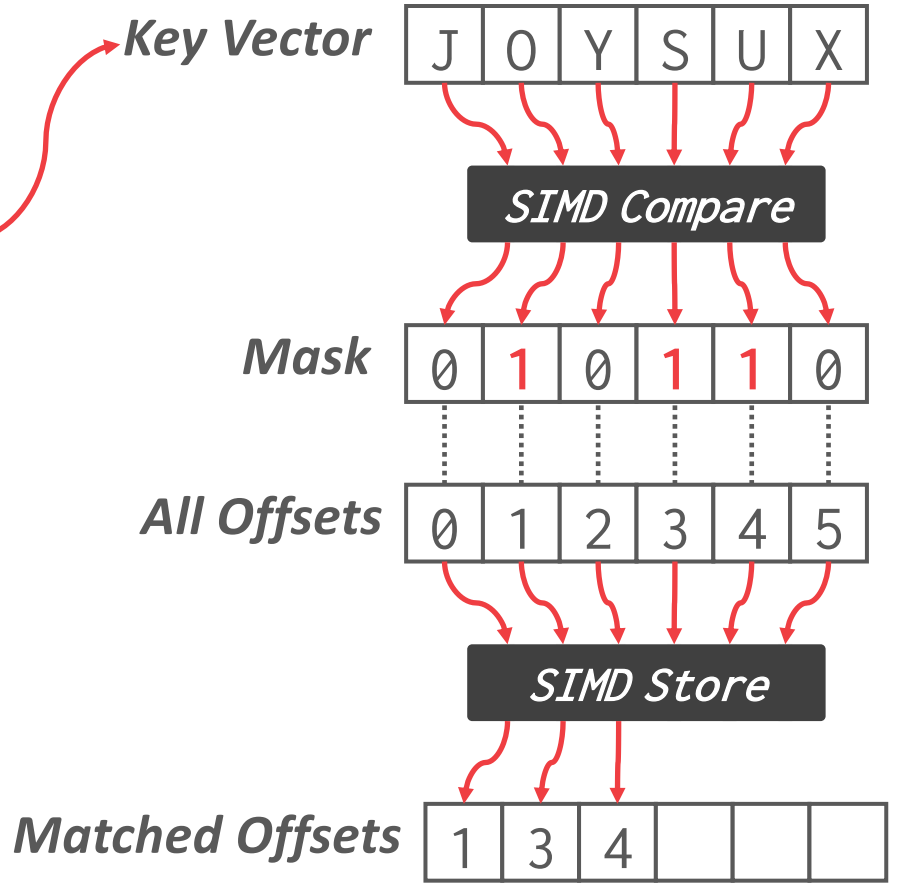
SELECTION SCANS

Vectorized

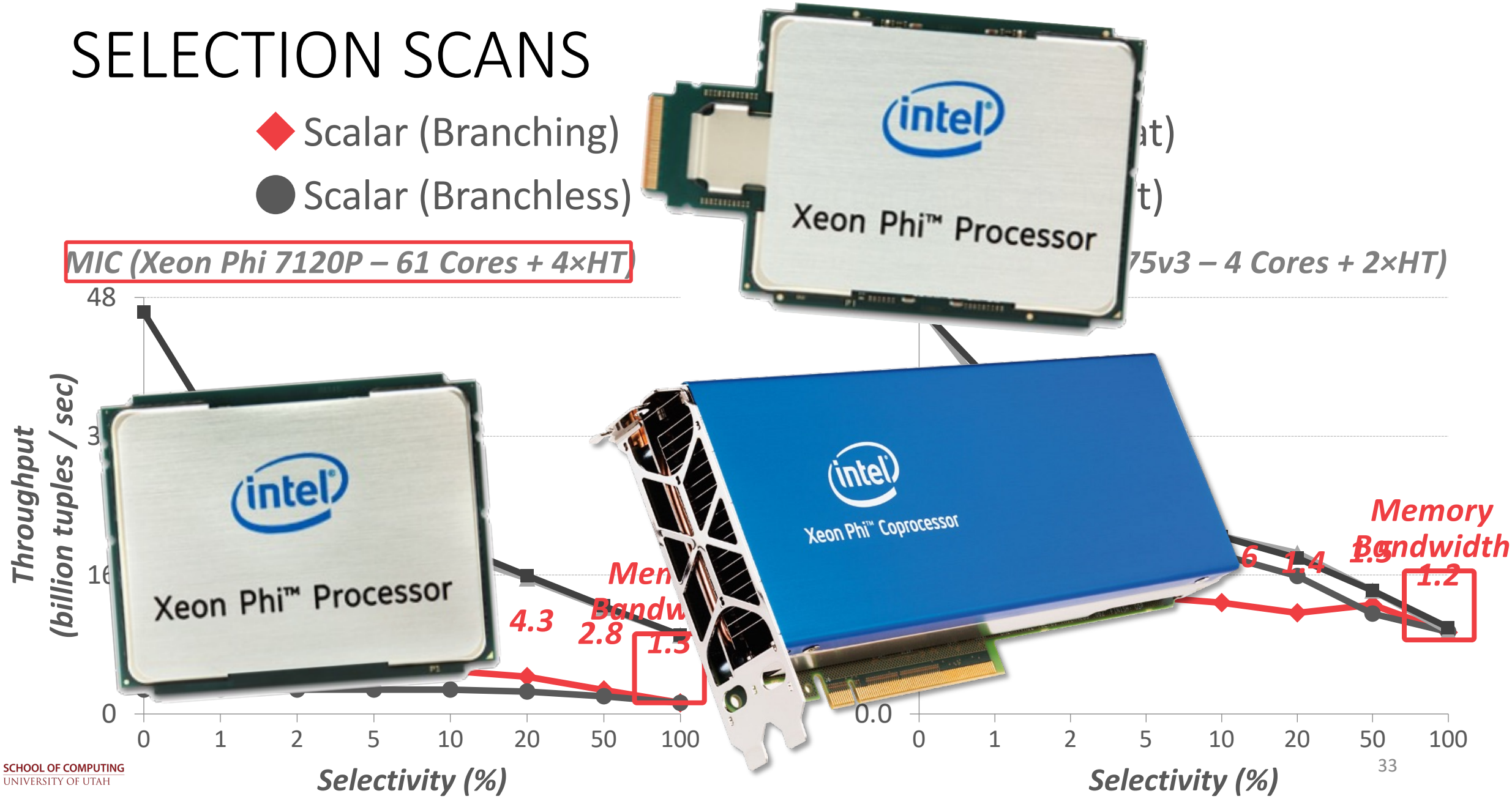
```
i = 0
for v_t in table:
    simdLoad(v_t.key, v_k)
    v_m = (v_k≥low ? 1 : 0) &
          (v_k≤high ? 1 : 0)
    simdStore(v_t, v_m, output[i])
    i = i + |v_m≠false|
```

```
SELECT * FROM table
 WHERE key >= "O" AND key <= "U"
```

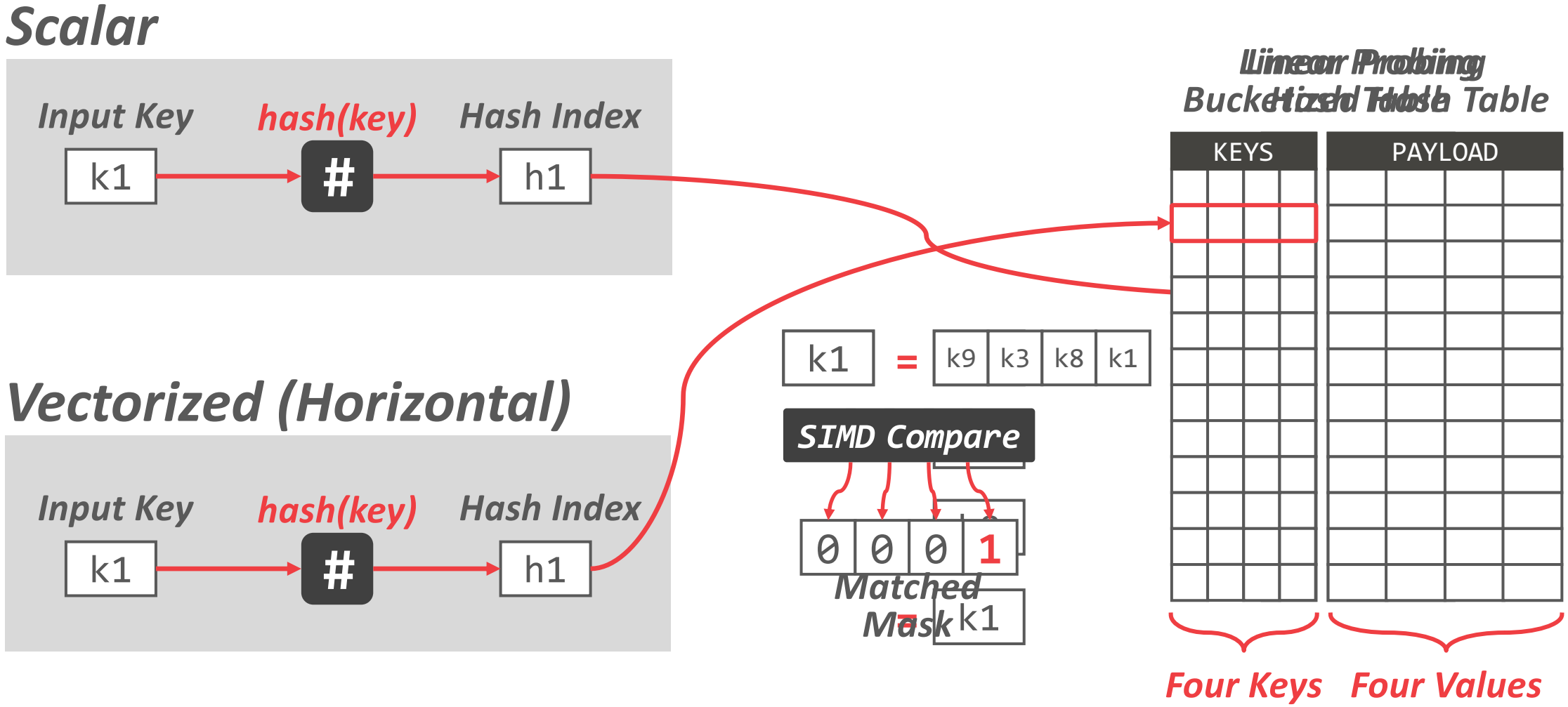
ID	KEY
1	J
2	O
3	Y
4	S
5	U
6	X



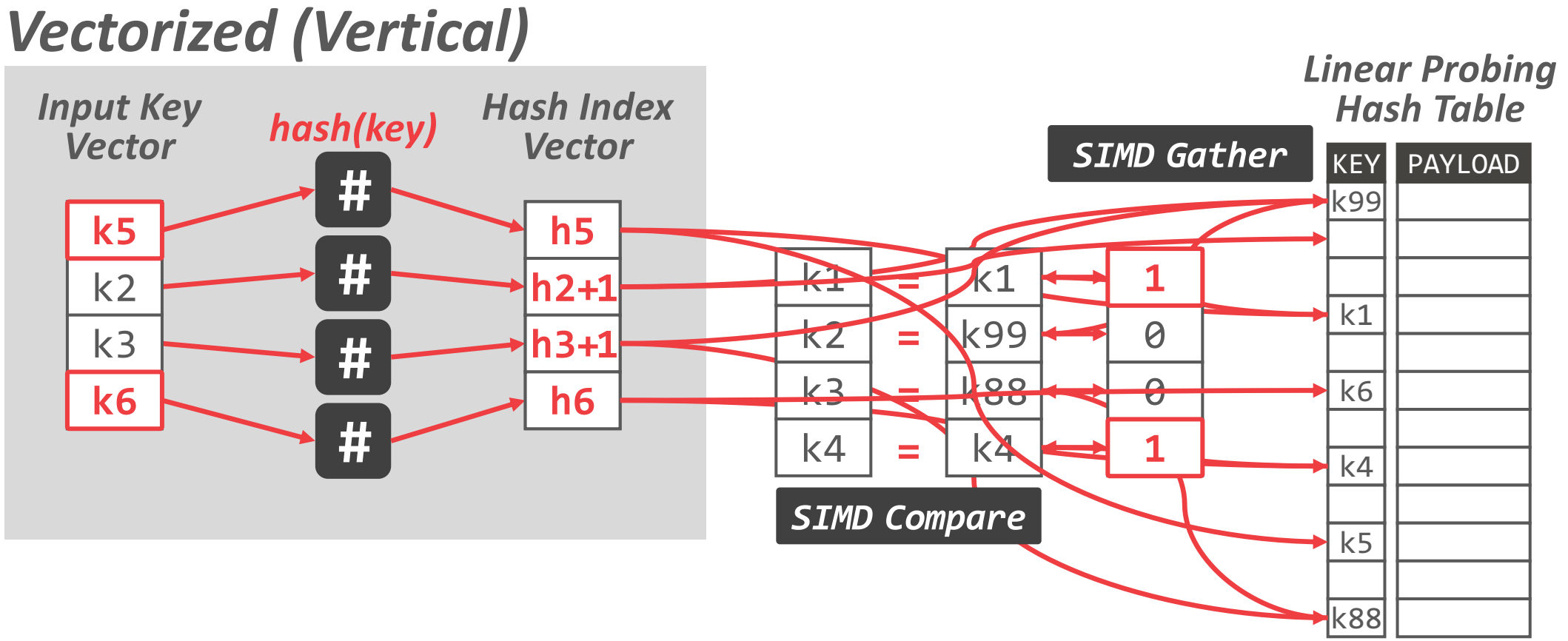
SELECTION SCANS



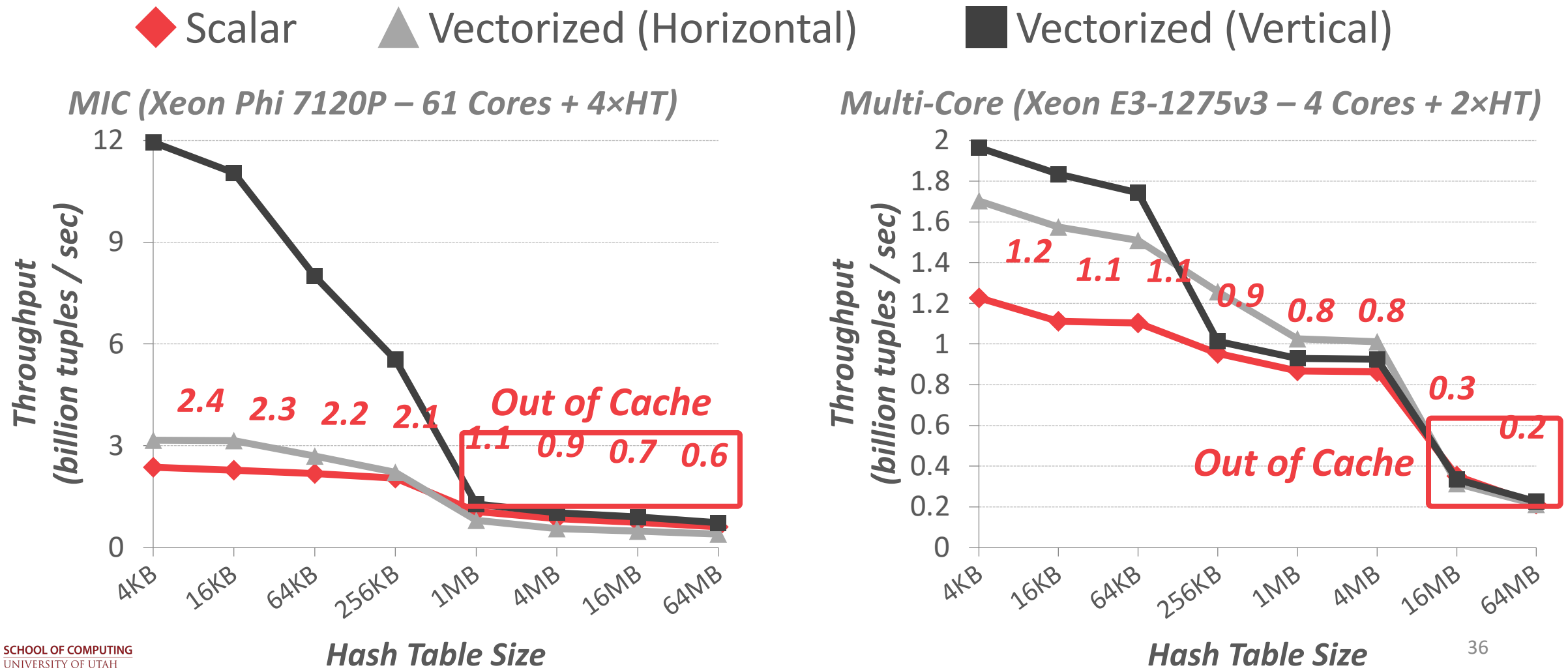
HASH TABLES – PROBING



HASH TABLES – PROBING

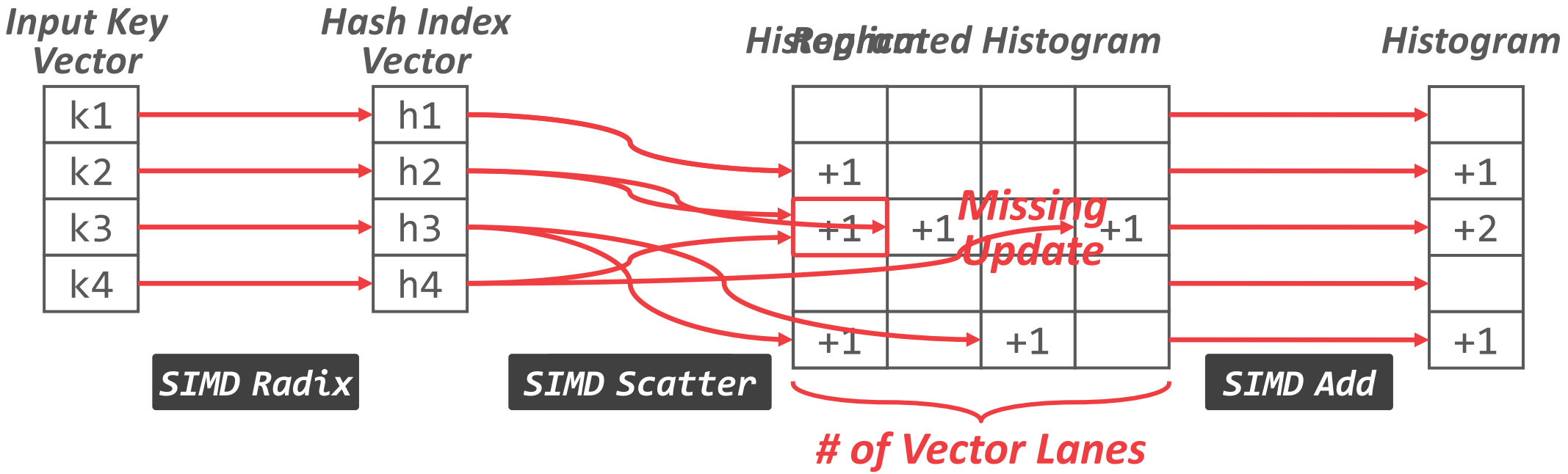


HASH TABLES – PROBING



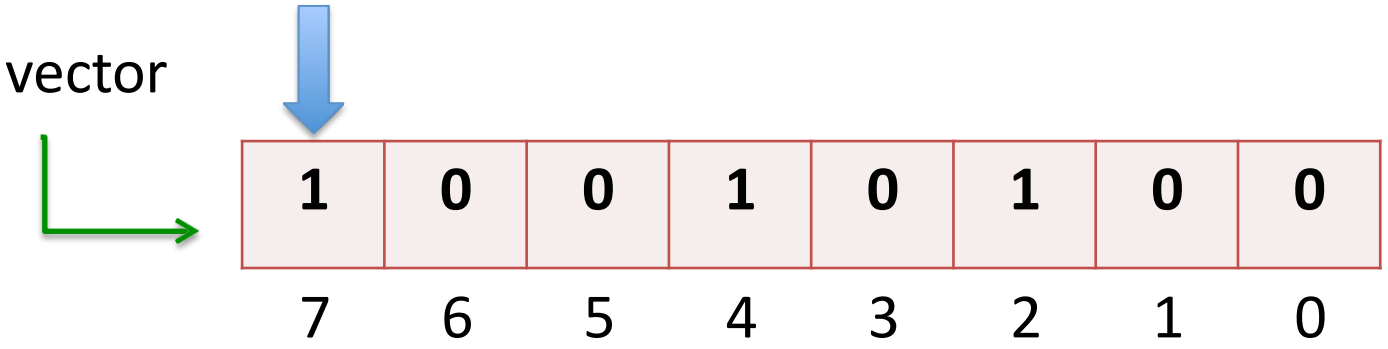
PARTITIONING – HISTOGRAM

- Use scatter and gathers to increment counts.
- Replicate the histogram to handle collisions.



Rank-Select Primitive

[Jacobson, Guy. "Space-efficient static trees and graphs." 1989]

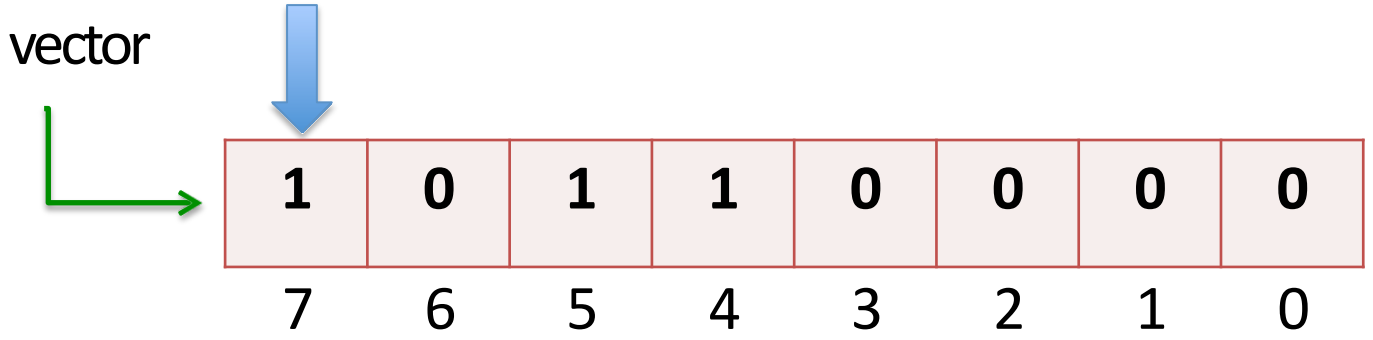


Rank(7) = 3

popcnt(vector) = 3

- The operation rank(i) is defined as the number of set bits in the vector..
- This is a generalization of *popcnt* instruction.
- *popcnt*: Bit set count.

Rank-Select Primitive



Select(3) = 7

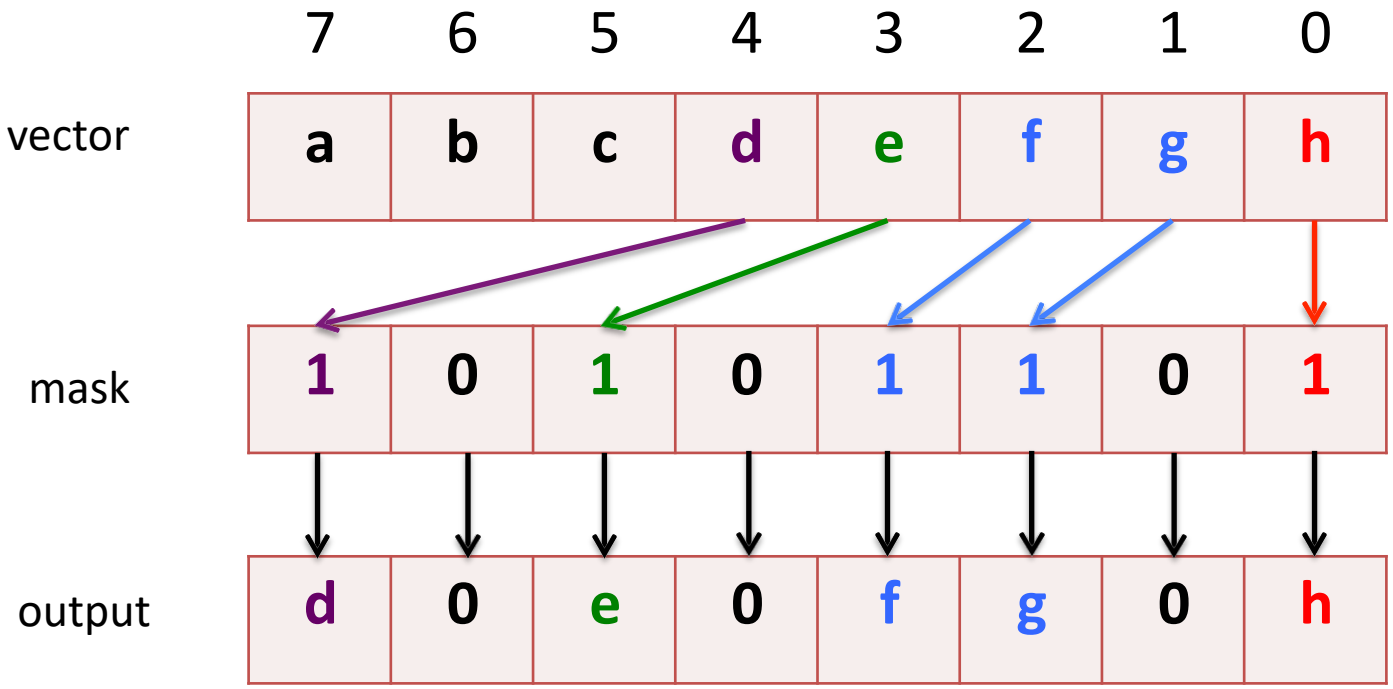
pdep(100,10110000) = 10000000

tzcnt(10000000) = 7

- The operation select(i) is defined as at which position is the ith set bit.
- We can implement select using *pdep* and *tzcnt* instructions.
- *pdep*: parallel bit deposit *tzcnt*: trailing zeros count.

PDEP Instruction

[Hilewitz, Yedidya, and Ruby B. Lee. "Fast bit compression and expansion with parallel extract and parallel deposit instructions."]



pdep: It copies the contiguous low-order bits to selected bits of the destination; other destination bits are cleared.