

# Lecture #20



# Carnegie Mellon University ADVANCED DATABASE SYSTEMS

## Vectorized Execution

@Andy\_Pavlo // 15-721 // Spring 2019

# TODAY'S AGENDA

---

Background

Hardware

Vectorized Algorithms (Columbia)



# 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**



# MULTI-CORE CPUS

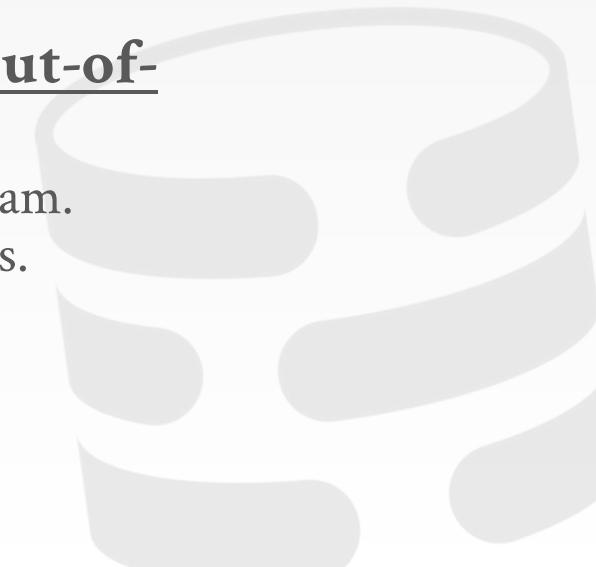
---

Use a small number of high-powered cores.

- Intel Xeon Skylake / Kaby Lake
- High power consumption and area per core.

Massively superscalar and aggressive out-of-order execution

- Instructions are issued from a sequential stream.
- Check for dependencies between instructions.
- Process multiple instructions per clock cycle.



# MANY INTEGRATED CORES (MIC)

Use a larger number of low-powered cores.

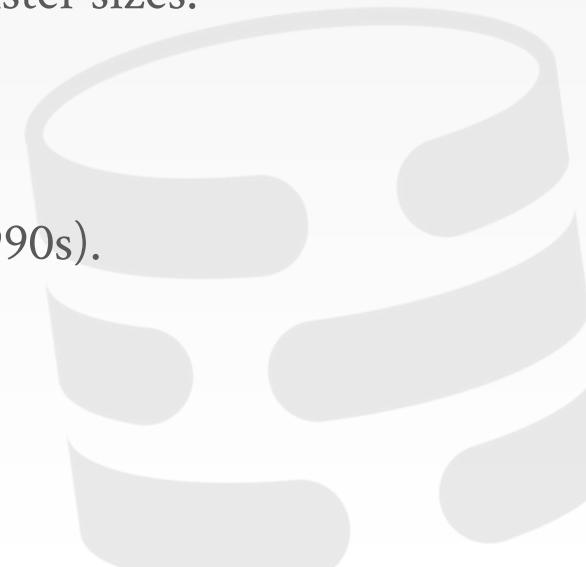
- Intel Xeon Phi
- Low power consumption and area per core.
- Expanded SIMD instructions with larger register sizes.

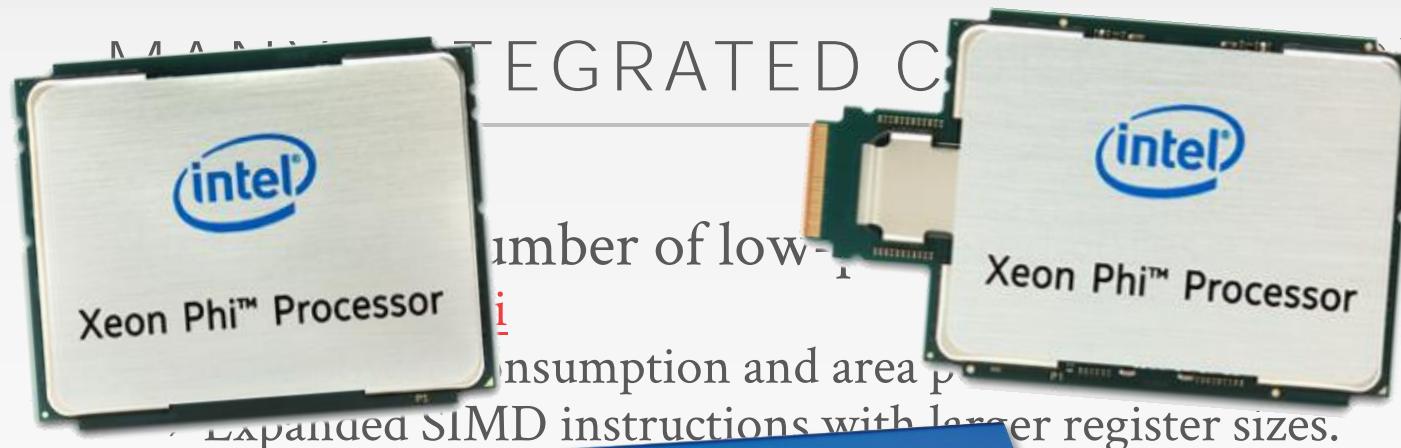
## Knights Ferry (Columbia Paper)

- Non-superscalar and in-order execution
- Cores = Intel P54C (aka Pentium from the 1990s).

## Knights Landing (Since 2016)

- Superscalar and out-of-order execution.
- Cores = Silvermont (aka Atom)





number of low-power cores  
consumption and area per core.  
Expanded SIMD instructions with larger register sizes.

## Knights

- Non-superscalar
- Cores = Itanium

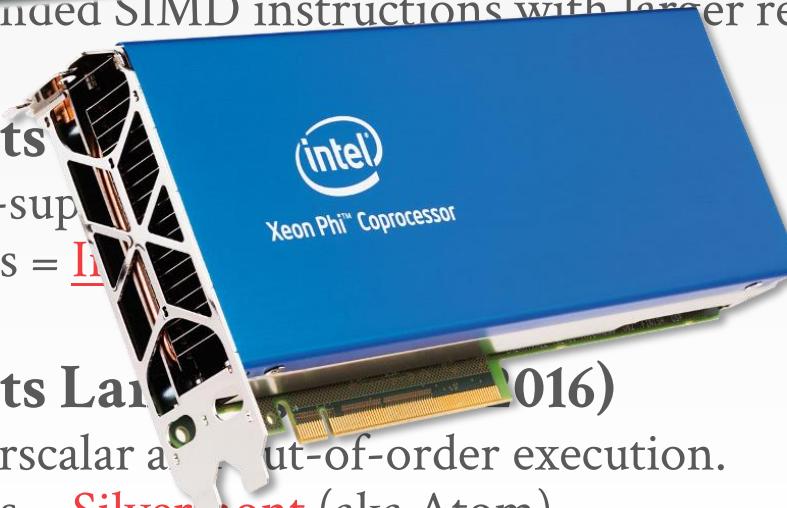


Xeon Phi™ Coprocessor

1990s).

## Knights Landing (2016)

- Superscalar architecture, out-of-order execution.
- Cores = Silvermont (aka Atom)



# 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

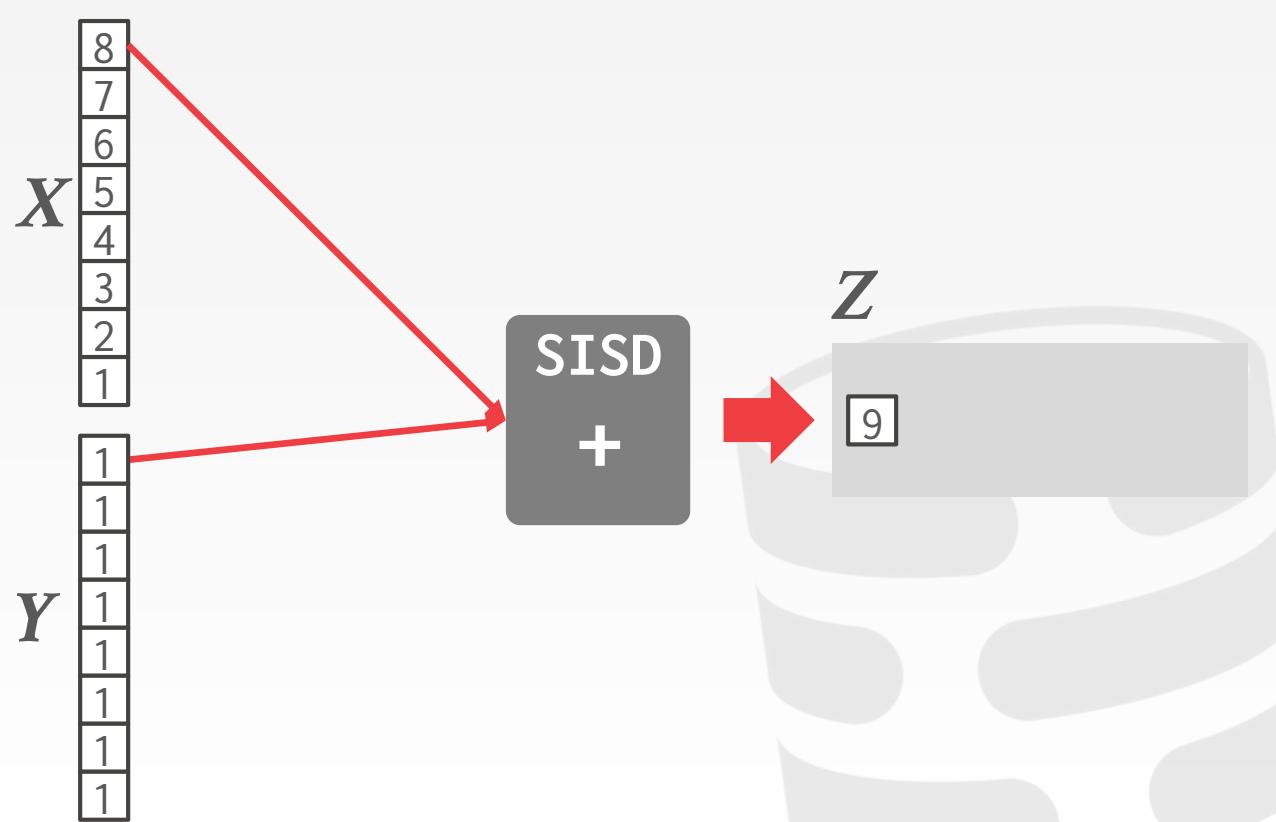


# SIMD EXAMPLE

$$X + Y = Z$$

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

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



# SIMD 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];  
}
```

X	8
	7
	6
	5
	4
	3
	2
	1

Y	1
	1
	1
	1
	1
	1
	1
	1

SISD  
+

Z

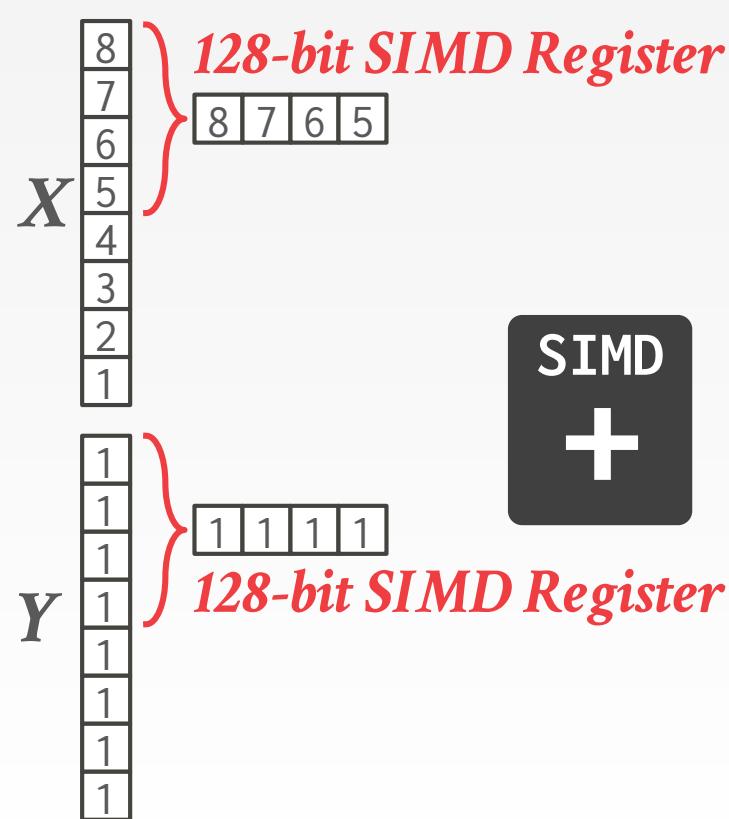
9	8	7	6	5	4	3	2
---	---	---	---	---	---	---	---

# 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];  
}
```

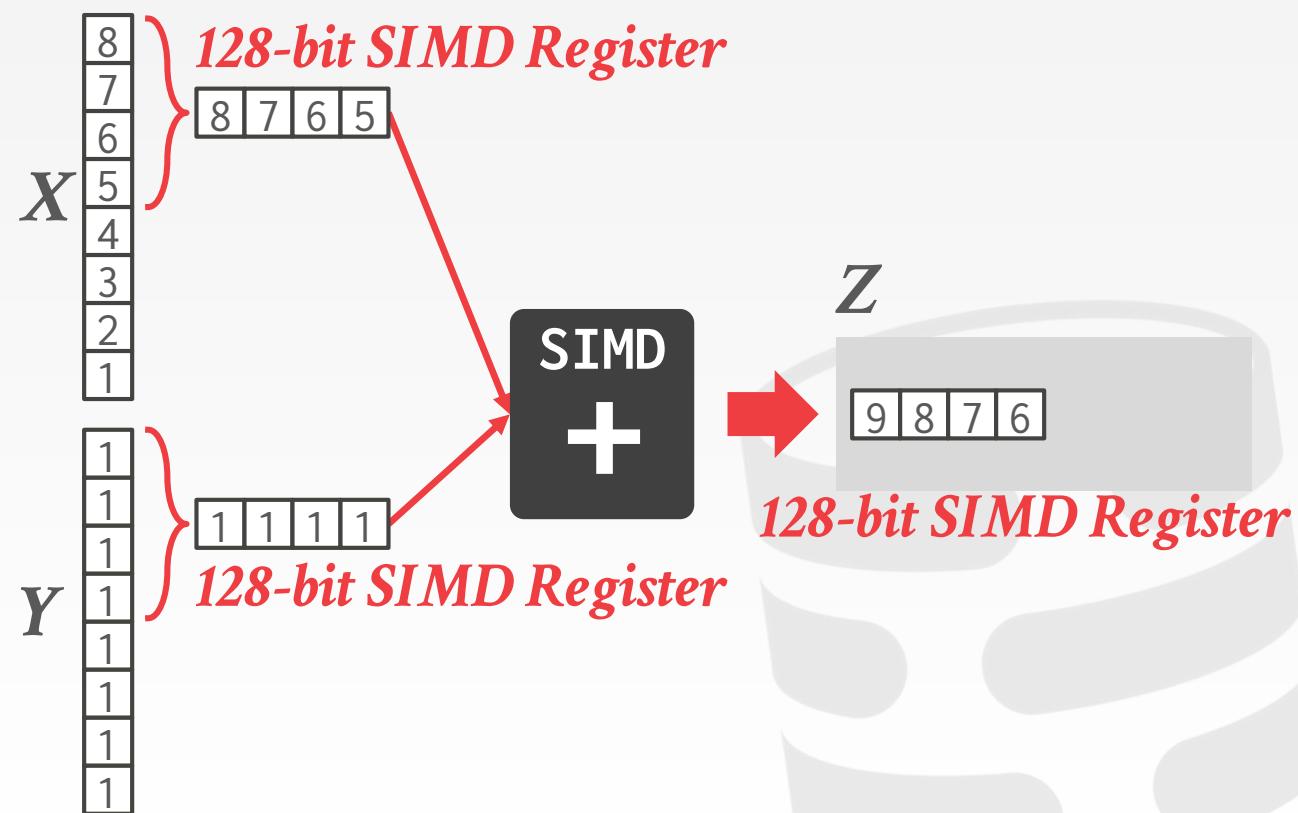


# SIMD EXAMPLE

$$X + Y = Z$$

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

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

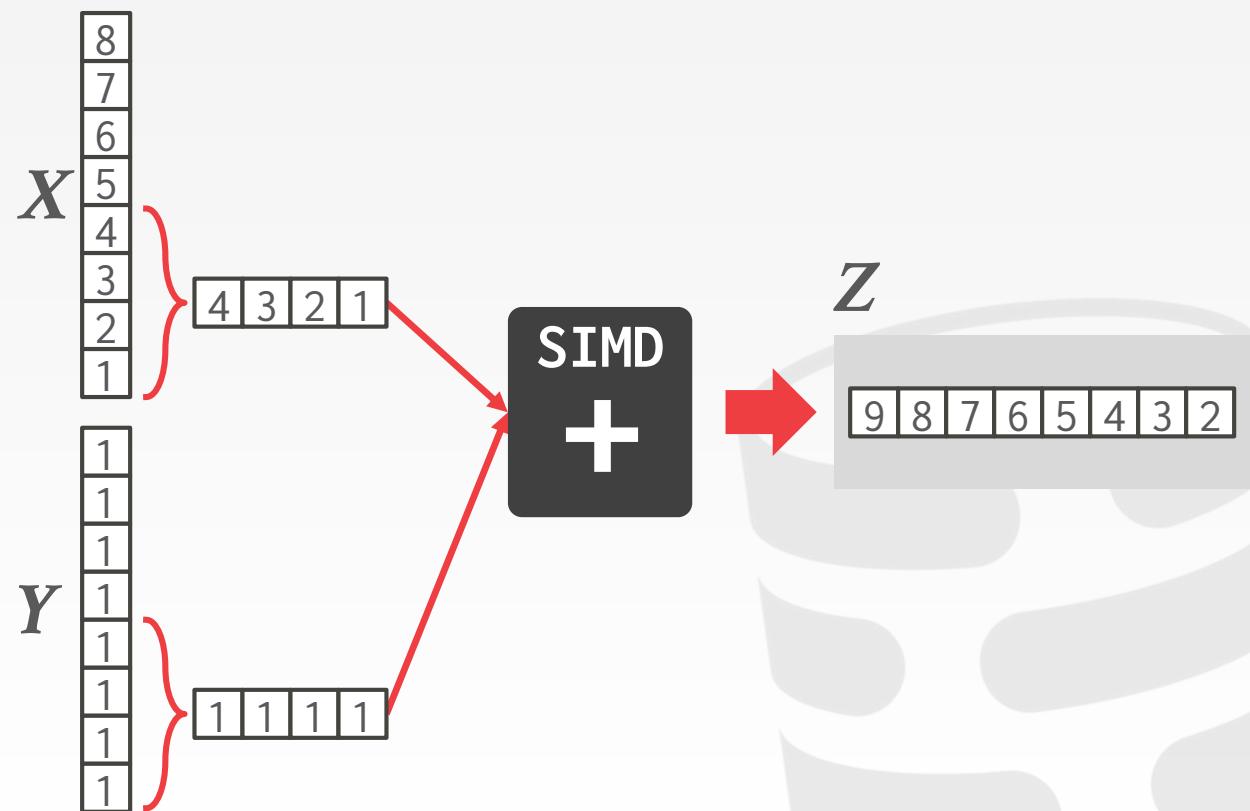


# SIMD EXAMPLE

$$X + Y = Z$$

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

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



# STREAMING SIMD EXTENSIONS (SSE)

SSE is a collection SIMD instructions that target special 128-bit SIMD registers.

These registers can be packed with four 32-bit scalars after which an operation can be performed on each of the four elements simultaneously.

First introduced by Intel in 1999.



# 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 ( $==, <, \leq, >, \geq, !=$ )

## 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

		<i>Width</i>	<i>Integers</i>	<i>Single-P</i>	<i>Double-P</i>
1997	MMX	64 bits	✓		
1999	SSE	128 bits	✓	✓ (x4)	
2001	SSE2	128 bits	✓	✓	✓ (x2)
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	✓	✓ (x8)	✓ (x4)
2013	AVX2	256 bits	✓	✓	✓
<b>2017</b>	<b>AVX-512</b>	<b>512 bits</b>	✓	✓ (x16)	✓ (x8)

Source: [James Reinders](#)

# VECTORIZATION

---

**Choice #1: Automatic Vectorization**

**Choice #2: Compiler Hints**

**Choice #3: Explicit Vectorization**



Source: [James Reinders](#)

# VECTORIZATION

Choice #1: Automatic Vectorization

Choice #2: Compiler Hints

Choice #3: Explicit Vectorization

*Ease of Use*

*Programmer Control*

# 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];  
    }  
}
```

This loop is not legal to automatically vectorize.



# AUTOMATIC VECTORIZATION

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

This loop is not legal to automatically vectorize.

*These might point to  
the same address!*



# AUTOMATIC VECTORIZATION

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

*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 as being done 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++));
    }
}
```

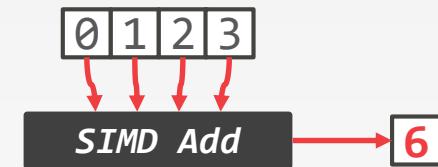
Store the vectors in 128-bit SIMD registers.

Then invoke the intrinsic to add together the vectors and write them to the output location.

# VECTORIZATION DIRECTION

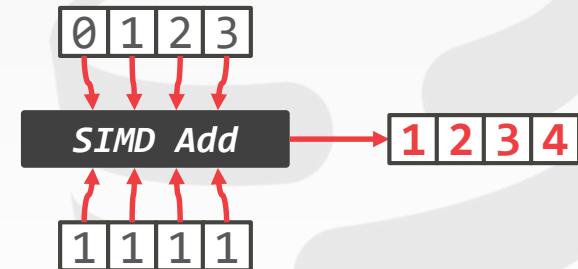
## Approach #1: Horizontal

- Perform operation on all elements together within a single vector.



## Approach #2: Vertical

- Perform operation in an elementwise manner on elements of each vector.



# 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 different things per lane subset.



# FUNDAMENTAL OPERATIONS

---

Selective Load

Selective Store

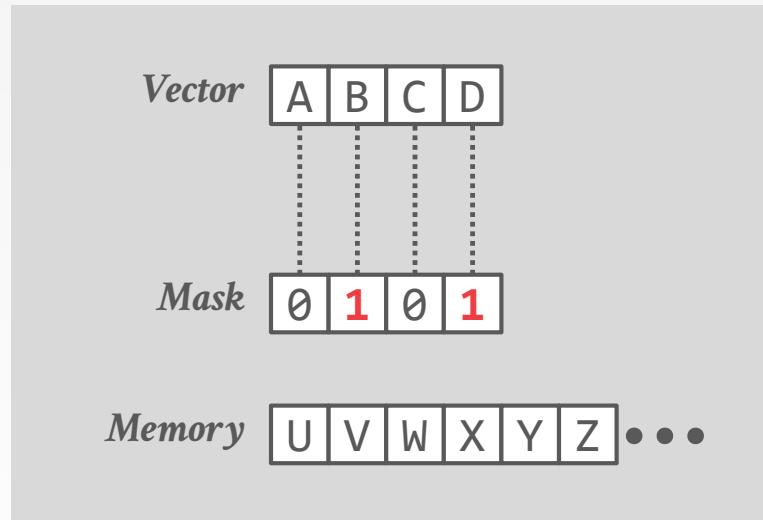
Selective Gather

Selective Scatter



# FUNDAMENTAL VECTOR OPERATIONS

## Selective Load



# FUNDAMENTAL VECTOR OPERATIONS

## Selective Load

Vector 

A	B	C	D
---	---	---	---

Mask 

0	1	0	1
---	---	---	---

Memory 

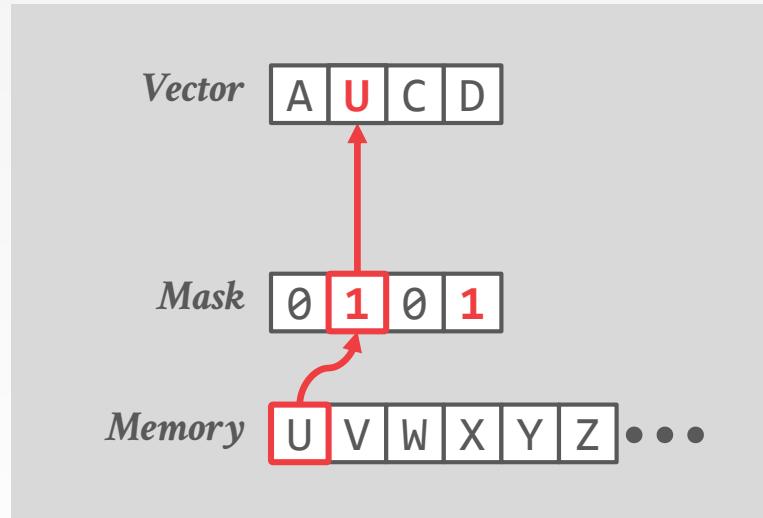
U	V	W	X	Y	Z
---	---	---	---	---	---

 • • •



# FUNDAMENTAL VECTOR OPERATIONS

## Selective Load



# FUNDAMENTAL VECTOR OPERATIONS

## Selective Load

Vector 

A	<b>U</b>	C	D
---	----------	---	---

Mask 

0	<b>1</b>	0	1
---	----------	---	---

Memory 

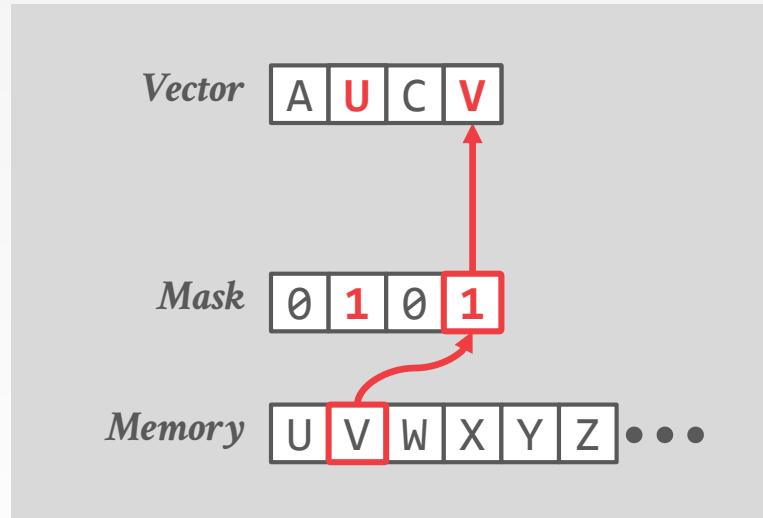
U	V	W	X	Y	Z
---	---	---	---	---	---

 • • •



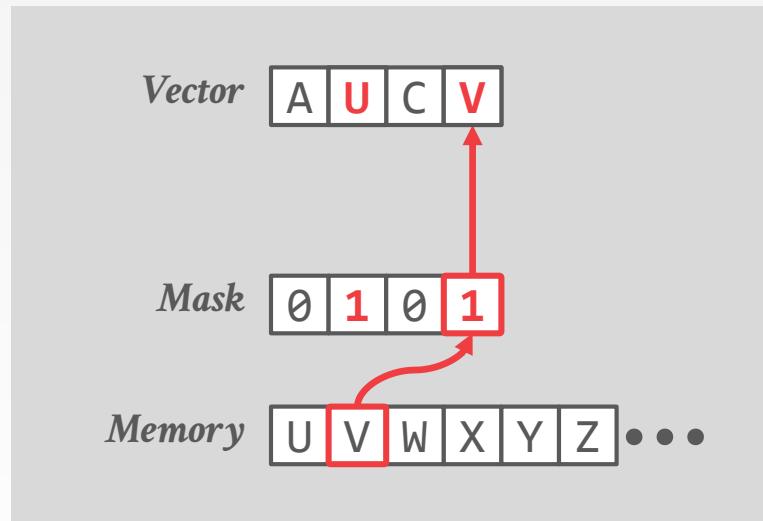
# FUNDAMENTAL VECTOR OPERATIONS

## Selective Load

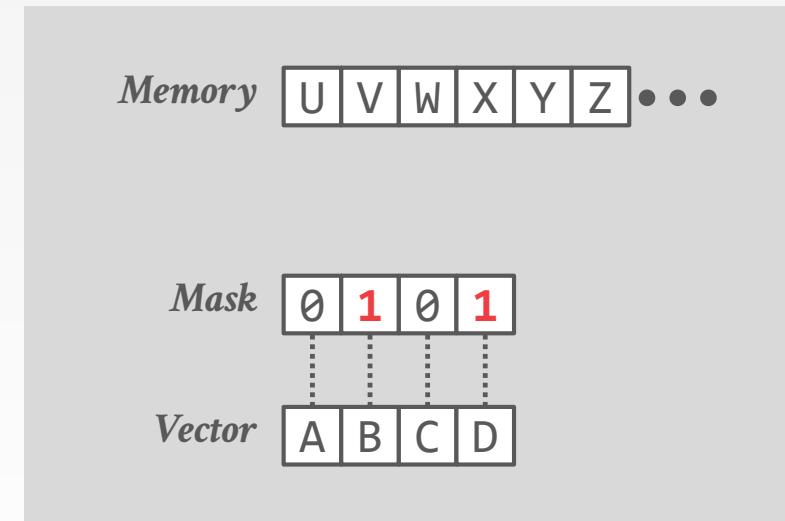


# FUNDAMENTAL VECTOR OPERATIONS

## Selective Load

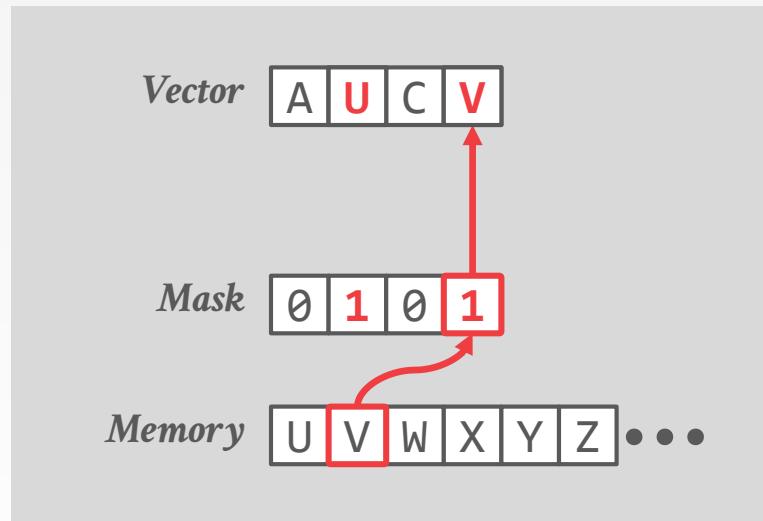


## Selective Store

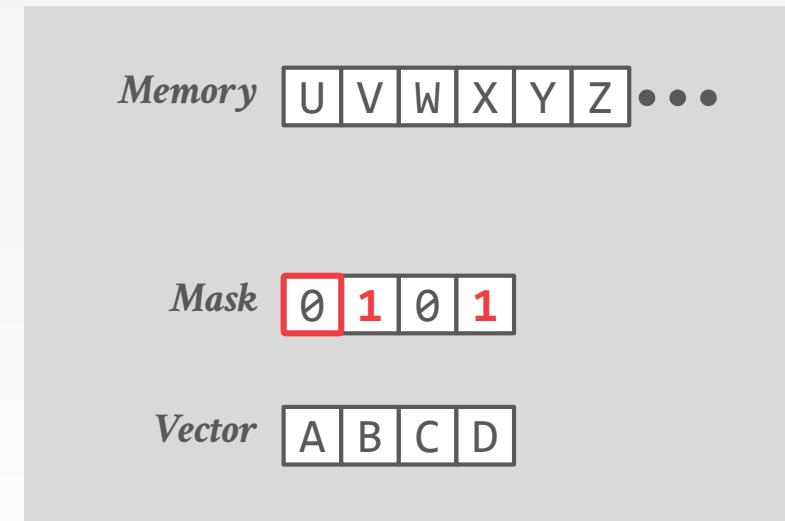


# FUNDAMENTAL VECTOR OPERATIONS

## Selective Load

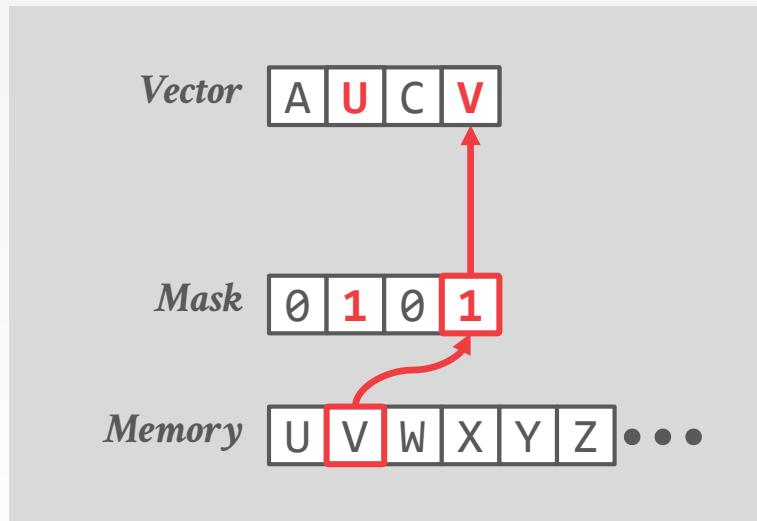


## Selective Store

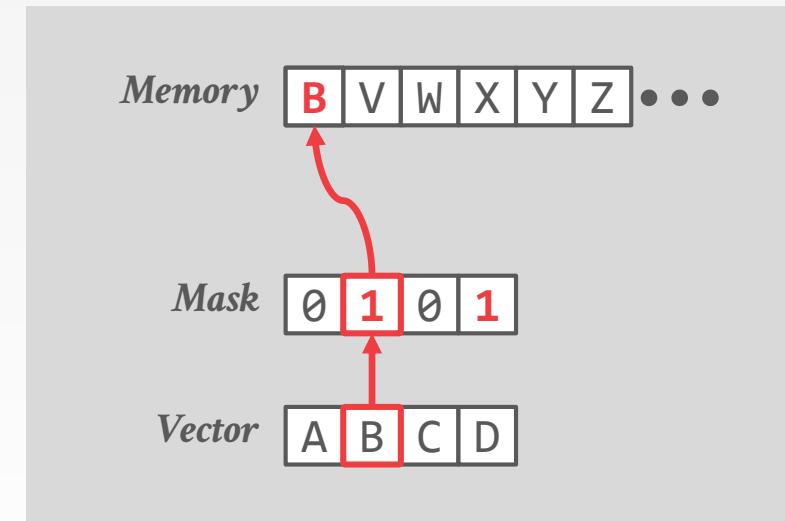


# FUNDAMENTAL VECTOR OPERATIONS

## Selective Load

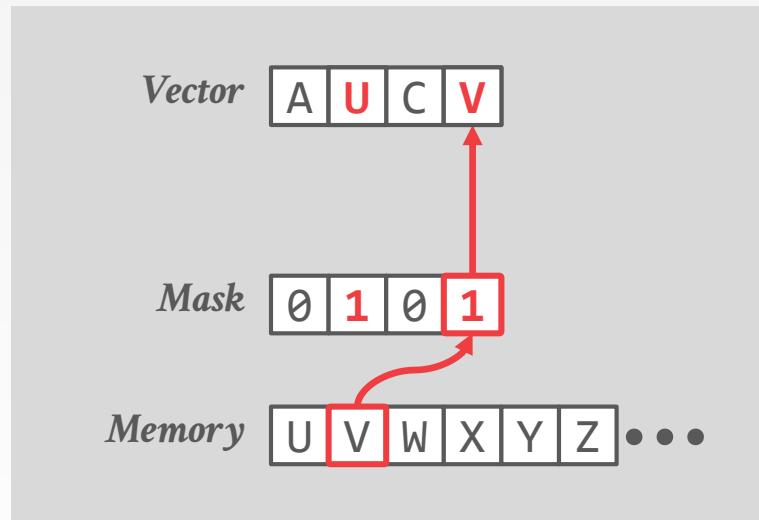


## Selective Store

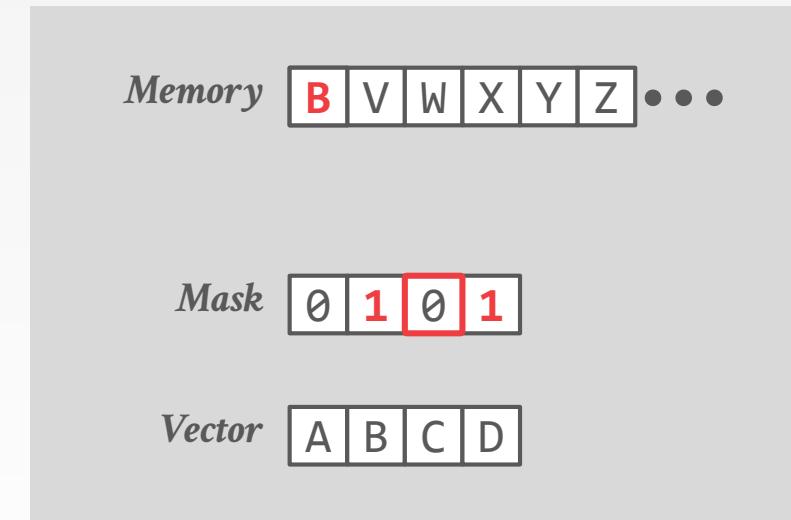


# FUNDAMENTAL VECTOR OPERATIONS

## Selective Load

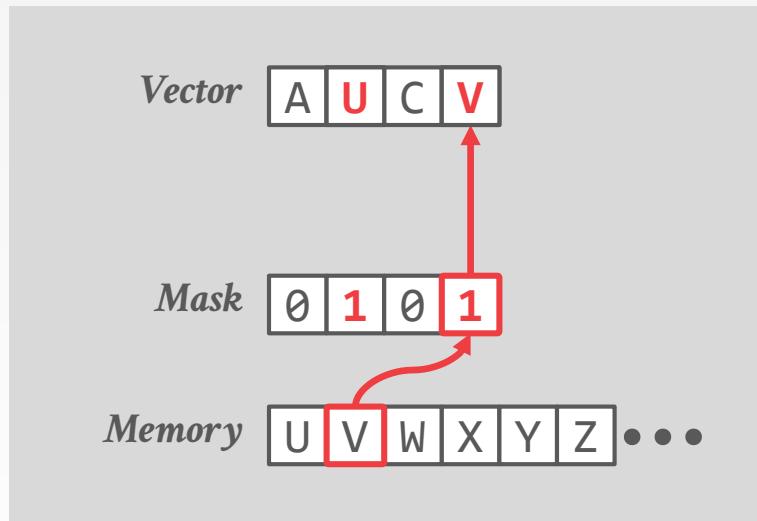


## Selective Store

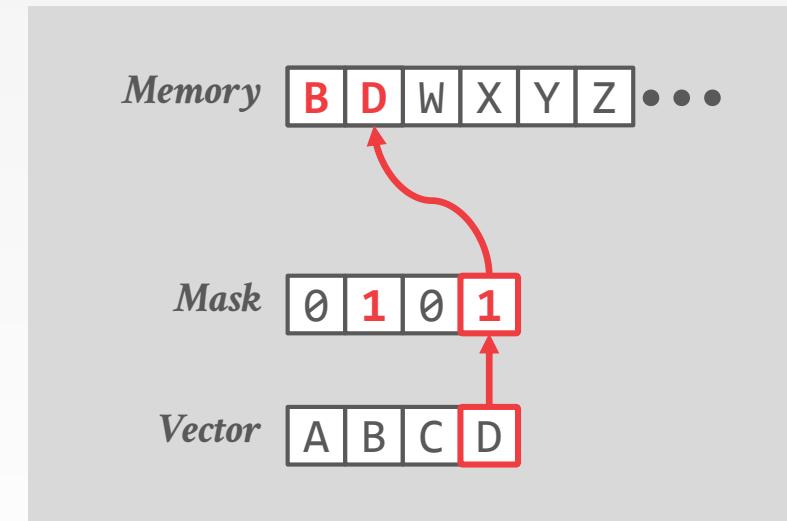


# FUNDAMENTAL VECTOR OPERATIONS

## Selective Load

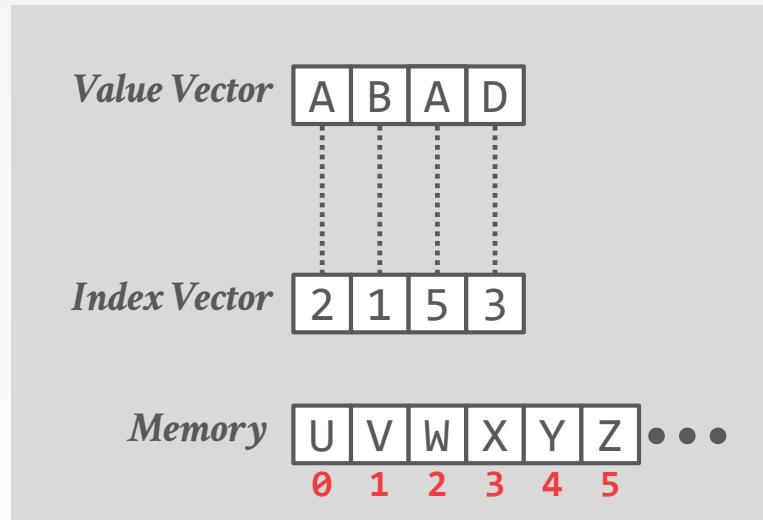


## Selective Store



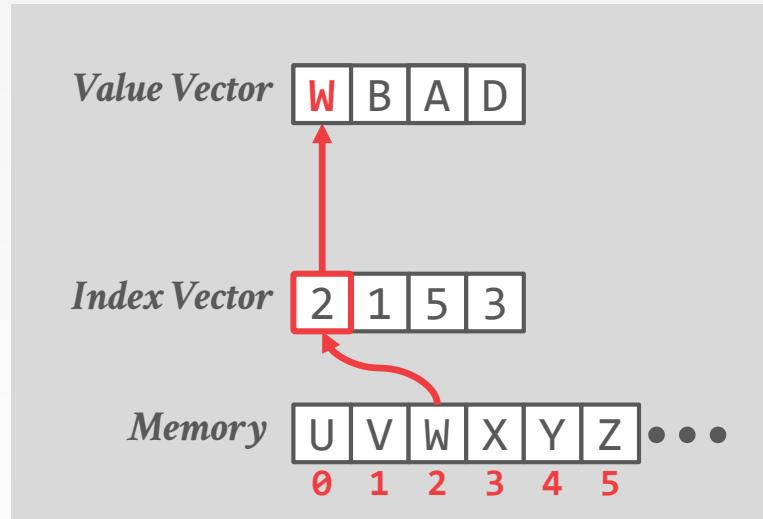
# FUNDAMENTAL VECTOR OPERATIONS

## Selective Gather



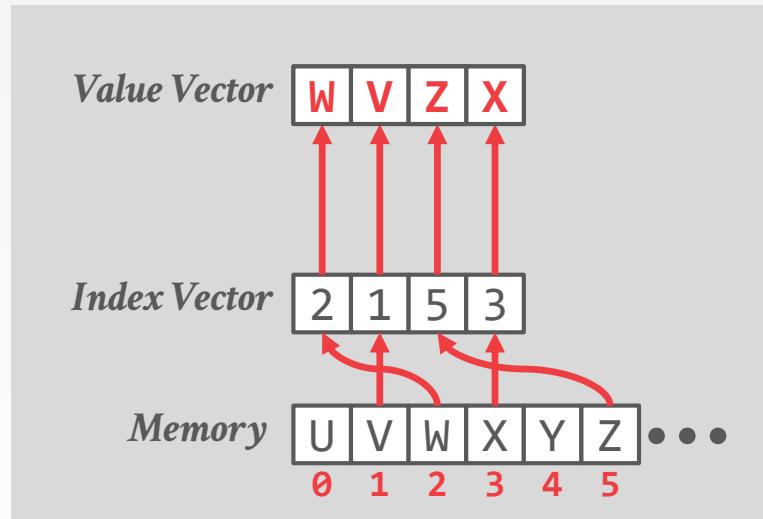
# FUNDAMENTAL VECTOR OPERATIONS

## Selective Gather



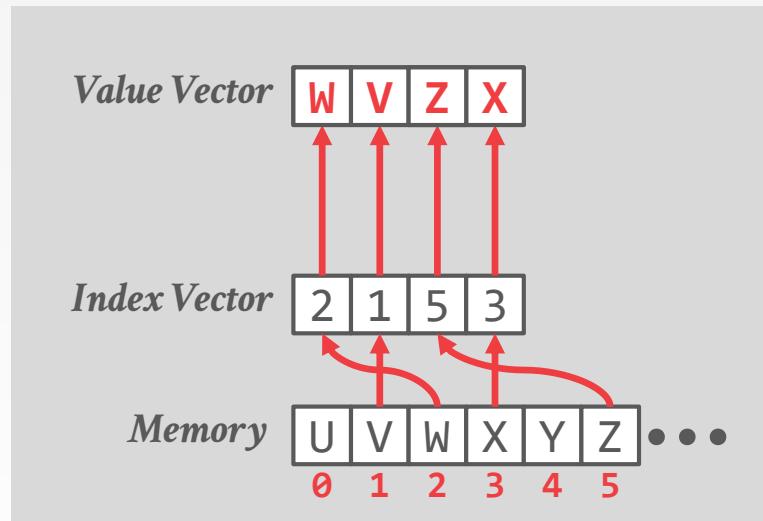
# FUNDAMENTAL VECTOR OPERATIONS

## Selective Gather

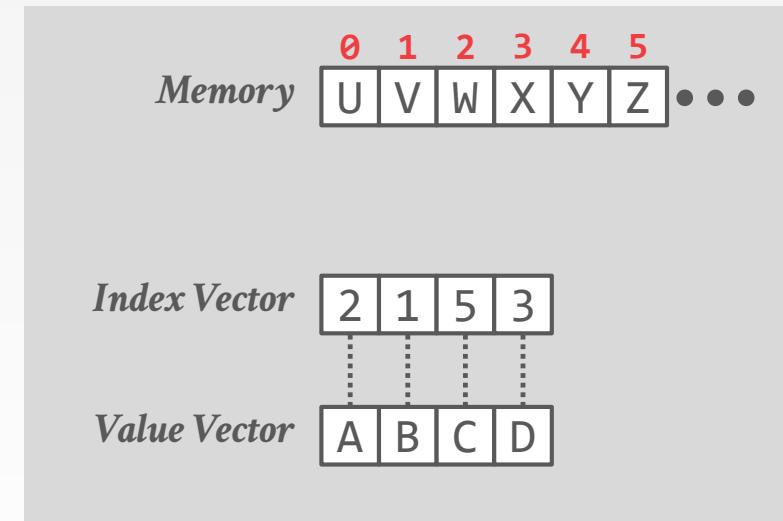


# FUNDAMENTAL VECTOR OPERATIONS

## Selective Gather

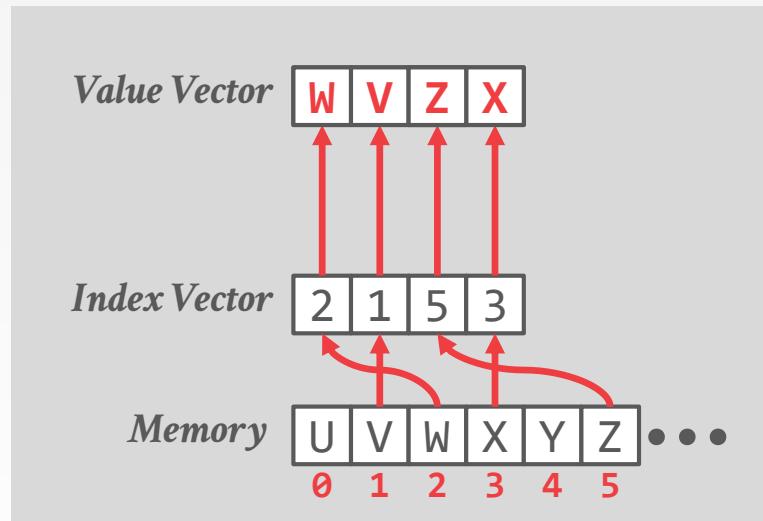


## Selective Scatter

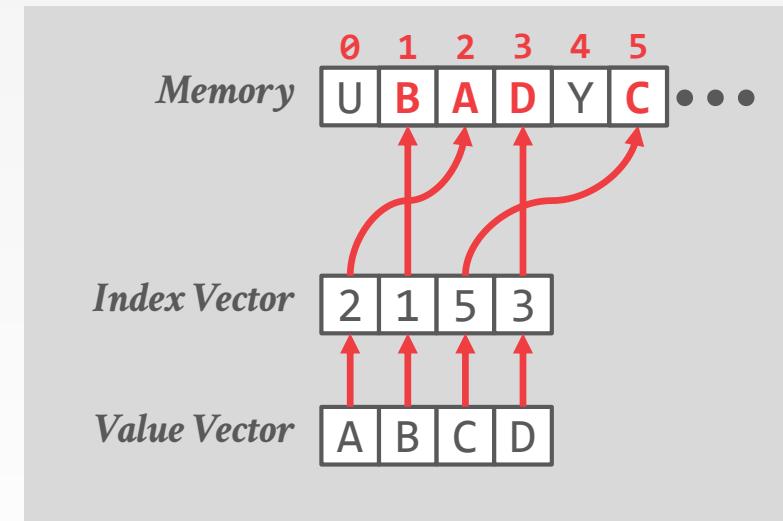


# FUNDAMENTAL VECTOR OPERATIONS

## 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

Paper provides additional info:  
→ 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
```

# SELECTION SCANS

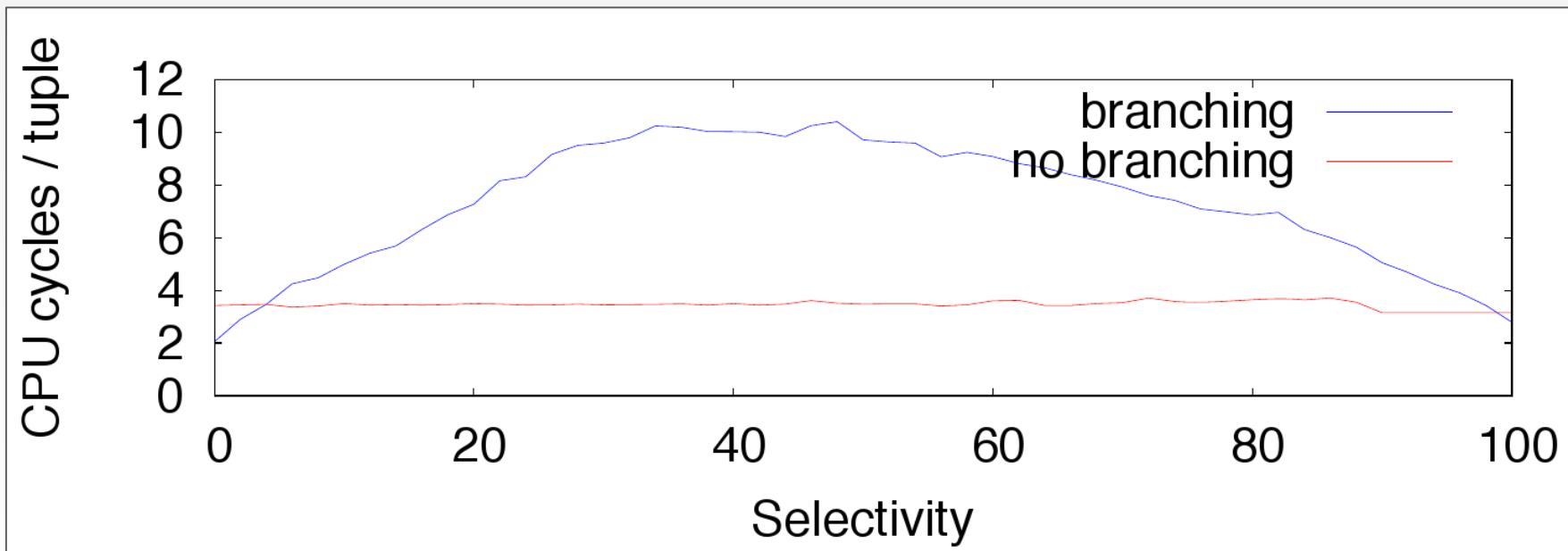
## Scalar (*Branching*)

```
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
```

# SELECTION SCANS



Source: [Bogdan Raducanu](#)

# SELECTION SCANS

## Vectorized

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



# SELECTION SCANS

## Vectorized

```
i = 0
for vt in table:
    SIMDLoad(vt.key, vk)
    vm = (vk ≥ low ? 1 : 0) &&
          (vk ≤ high ? 1 : 0)
    SIMDStore(vt, vm, output[i])
    i = i + |vm ≠ false|
```

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

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



# SELECTION SCANS

## Vectorized

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

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

Key Vector

J	O	Y	S	U	X
---	---	---	---	---	---

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



# SELECTION SCANS

## Vectorized

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

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

Key Vector

J	O	Y	S	U	X
---	---	---	---	---	---

SIMD Compare

Mask

0	1	0	1	1	0
---	---	---	---	---	---

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

# SELECTION SCANS

## Vectorized

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

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

*Key Vector*

J	O	Y	S	U	X
---	---	---	---	---	---

*Mask*

0	1	0	1	1	0
---	---	---	---	---	---

*All Offsets*

0	1	2	3	4	5
---	---	---	---	---	---

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

# SELECTION SCANS

## Vectorized

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

```
SELECT * FROM table
WHERE key >= "0" AND key <= "U"
```

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

Key Vector

J	O	Y	S	U	X
---	---	---	---	---	---

Mask

0	1	0	1	1	0
---	---	---	---	---	---

All Offsets

0	1	2	3	4	5
---	---	---	---	---	---

Matched Offsets

1	3	4			
---	---	---	--	--	--

SIMD Compare

SIMD Store

## Selection Scans (Vectorized)

- ▶  $i = 0$
- ▶ For  $V_t$  in table:
  - ▶ `simdLoad(Vt.key, Vk)`
  - ▶  $V_m = (V_k \geq "O"?1:0) \&& (V_k \leq "U"?1:0)$
  - ▶ If( $V_m \neq \text{false}$ ):
    - ▶ `simdStore(Vt, Vm, output[i])`
    - ▶  $i = i + |V_m| \neq \text{false}|$

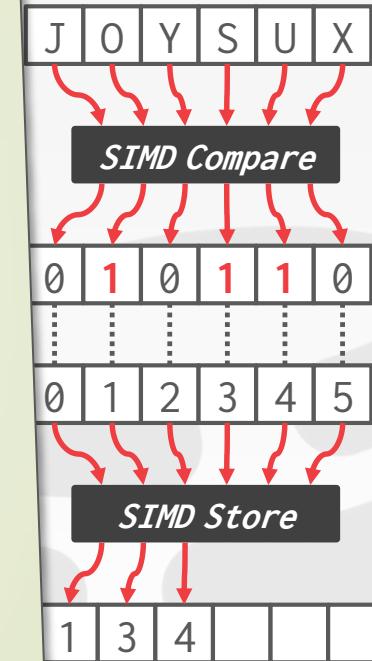
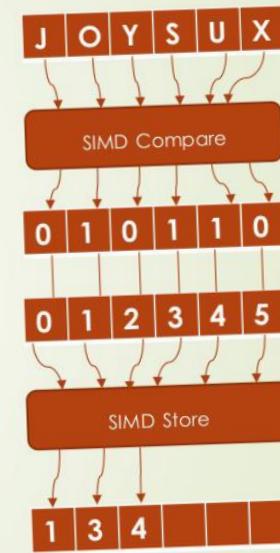
Key Vector

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

Mask

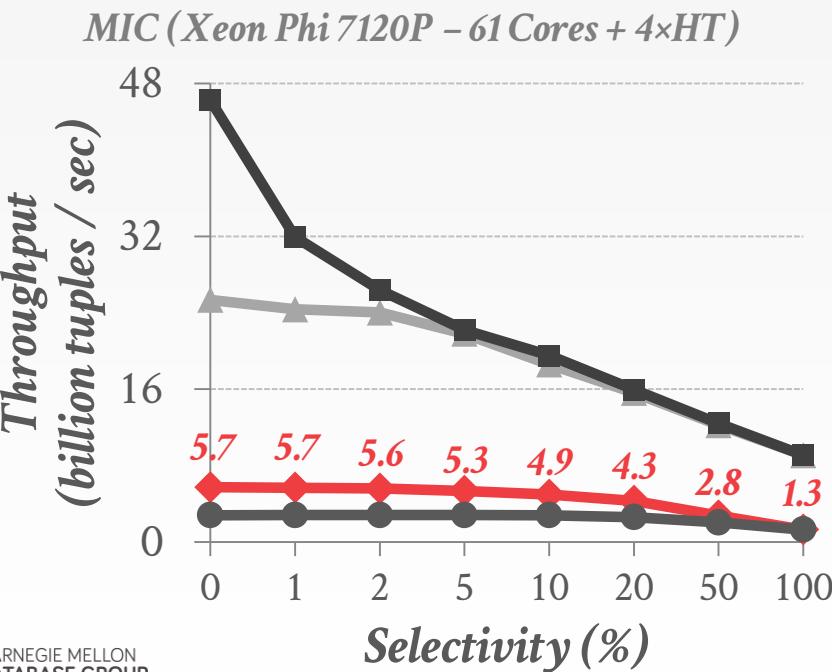
Matched  
Offsets

All Offsets

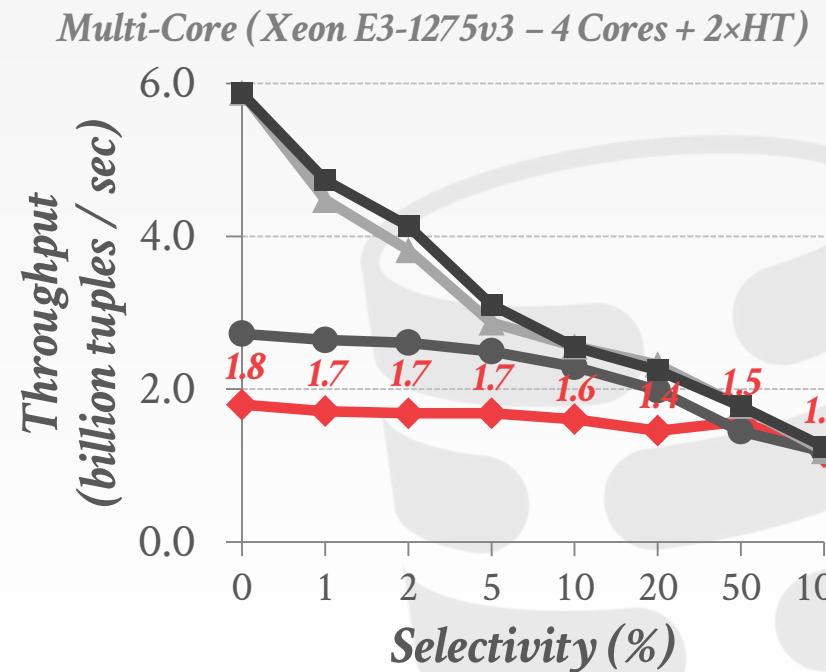


# SELECTION SCANS

- ◆ Scalar (Branching)
- Scalar (Branchless)



- ▲ Vectorized (Early Mat)
- Vectorized (Late Mat)

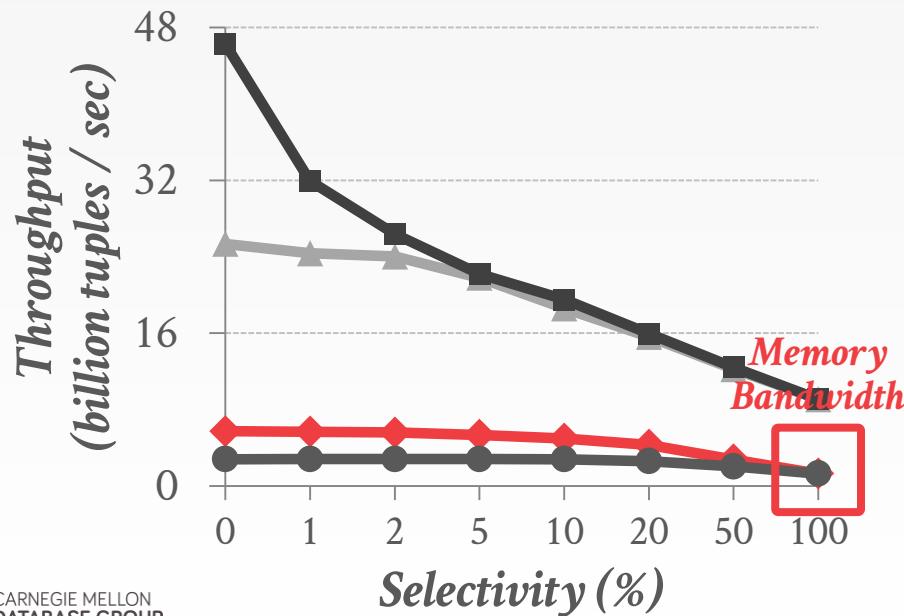


# SELECTION SCANS

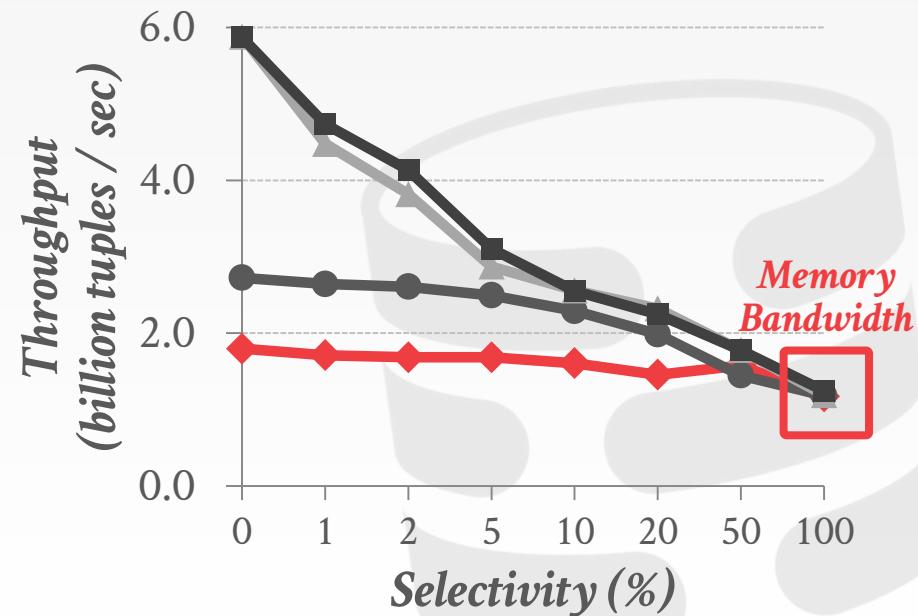
- ◆ Scalar (Branching)
- Scalar (Branchless)

- ▲ Vectorized (Early Mat)
- Vectorized (Late Mat)

*MIC (Xeon Phi 7120P – 61 Cores + 4×HT)*

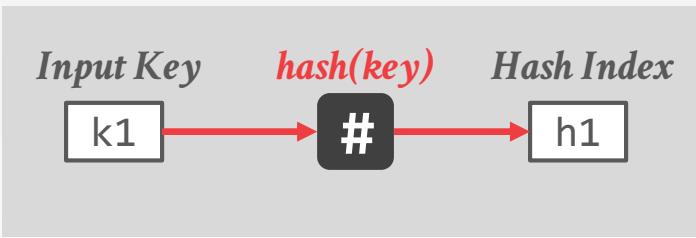


*Multi-Core (Xeon E3-1275v3 – 4 Cores + 2×HT)*



# HASH TABLES – PROBING

## Scalar

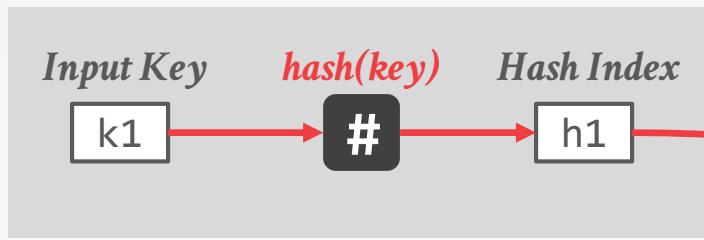


*Linear Probing  
Hash Table*

KEY	PAYLOAD

# HASH TABLES – PROBING

## Scalar



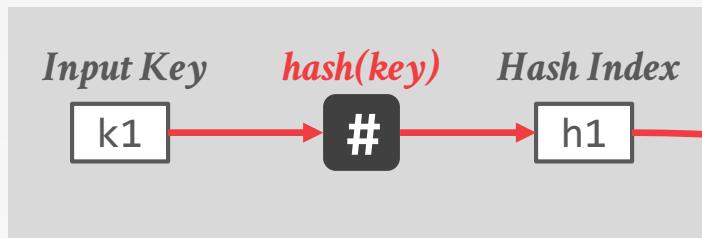
$$k_1 = k_9$$

*Linear Probing  
Hash Table*

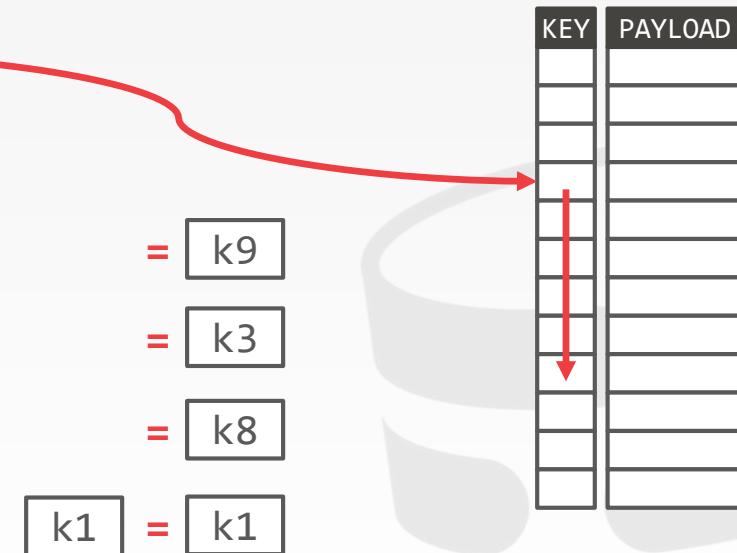
KEY	PAYLOAD

# HASH TABLES – PROBING

## Scalar

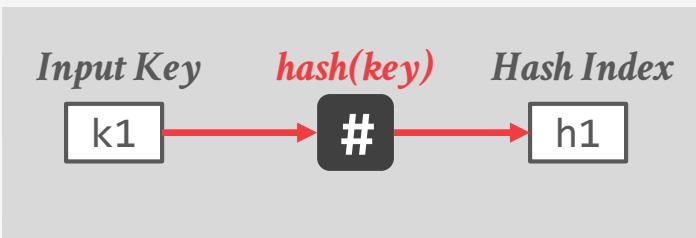


*Linear Probing Hash Table*

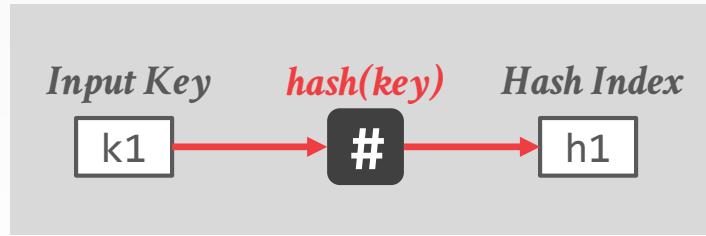


# HASH TABLES – PROBING

## Scalar



## Vectorized (Horizontal)

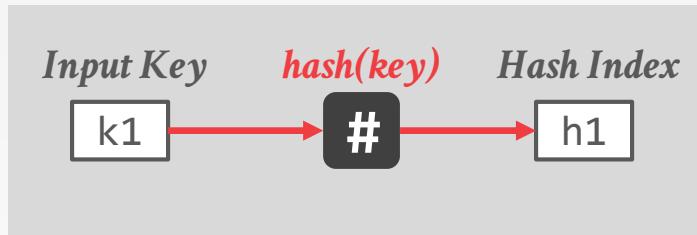


Linear Probing  
Bucketized Hash Table

KEYS	PAYLOAD

# HASH TABLES – PROBING

## Scalar

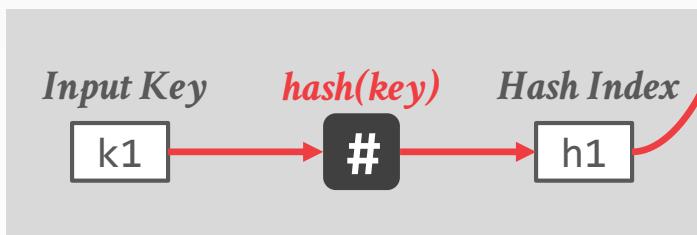


*Linear Probing  
Bucketized Hash Table*

KEYS	PAYLOAD
k9 k3 k8 k1	

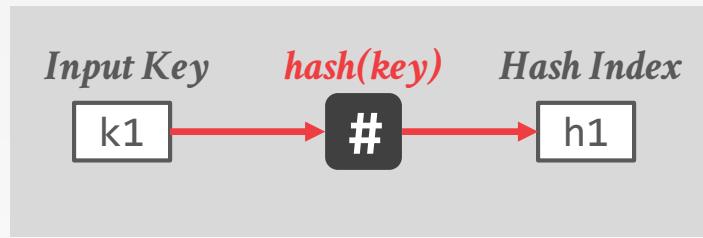
$k1 = [k9, k3, k8, k1]$

## Vectorized (Horizontal)

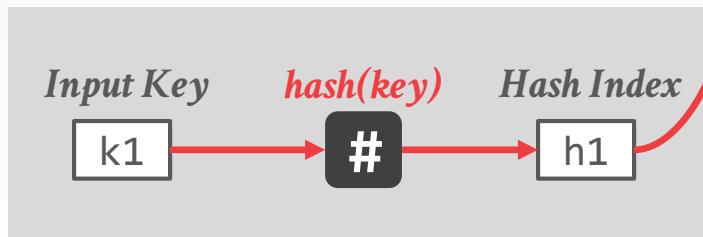


# HASH TABLES – PROBING

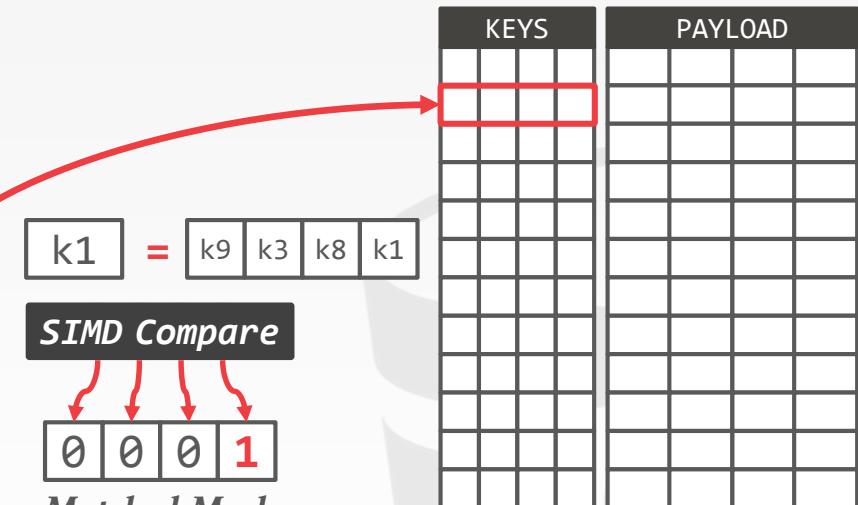
## Scalar



## Vectorized (Horizontal)



*Linear Probing Bucketized Hash Table*



# HASH TABLES – PROBING

## *Vectorized (Vertical)*

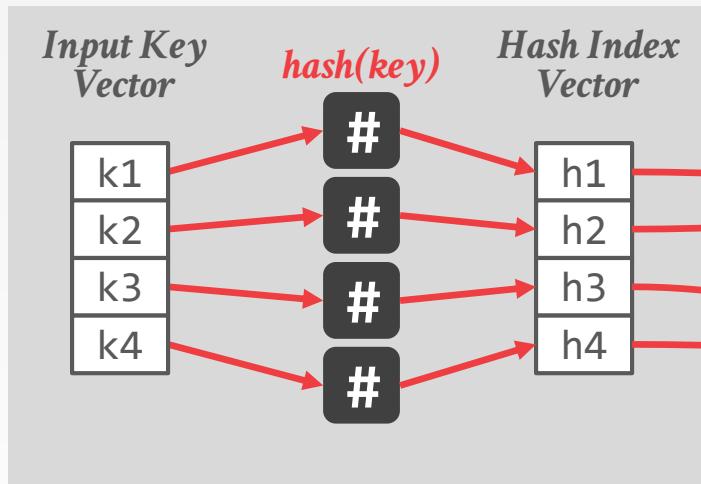
<i>Input Key Vector</i>
k1
k2
k3
k4

*Linear Probing  
Hash Table*

KEY	PAYLOAD
k99	
k1	
k6	
k4	
k5	
k88	

# HASH TABLES – PROBING

## *Vectorized (Vertical)*

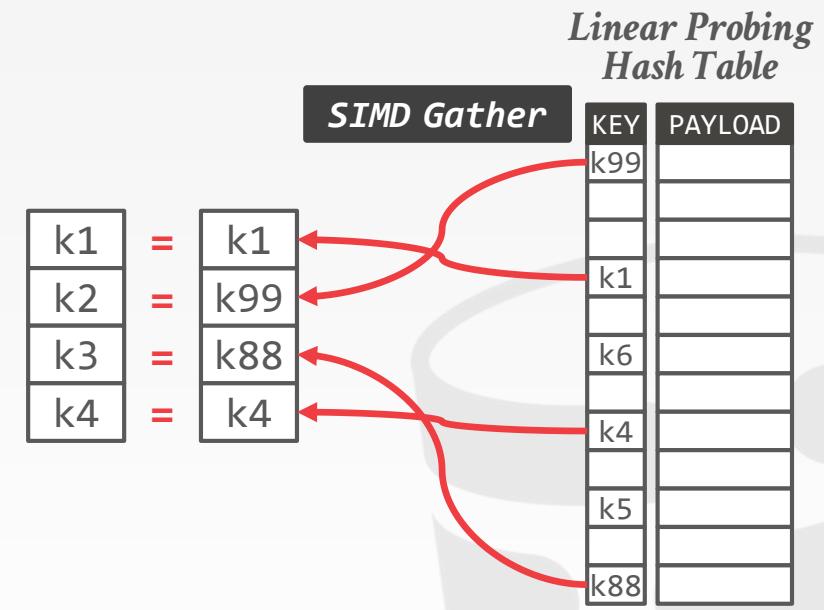
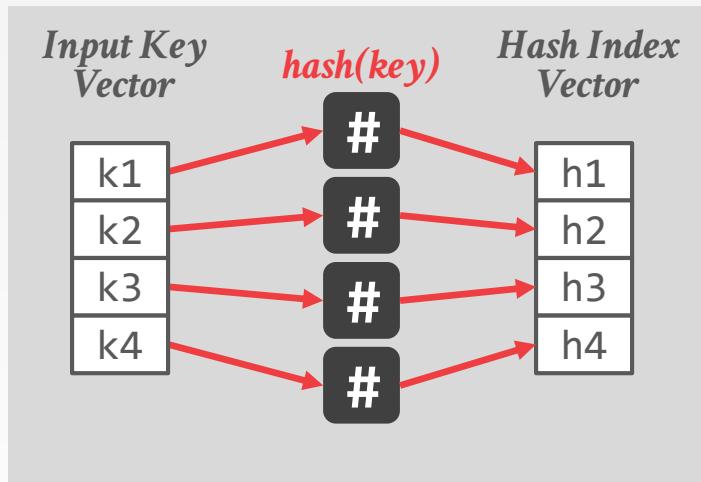


*Linear Probing Hash Table*

KEY	PAYLOAD
k99	
k1	
k6	
k4	
k5	
k88	

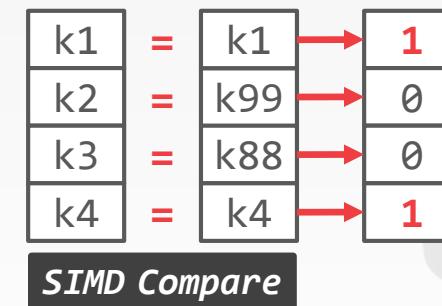
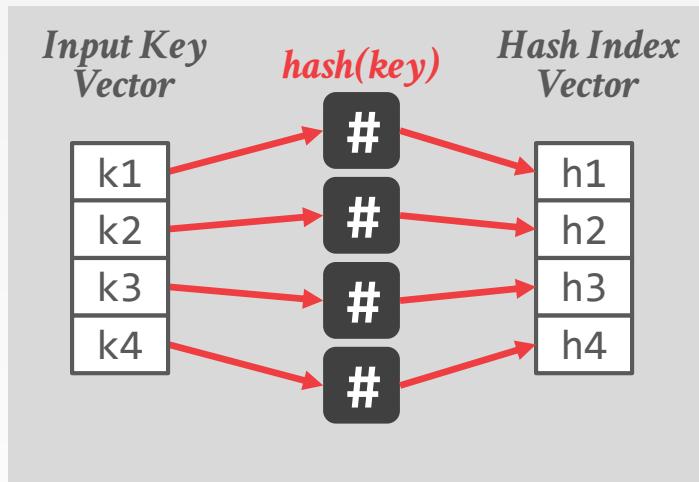
# HASH TABLES – PROBING

## *Vectorized (Vertical)*



# HASH TABLES – PROBING

## *Vectorized (Vertical)*

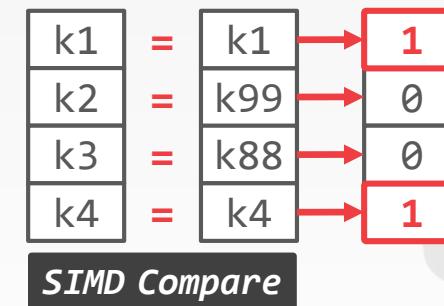
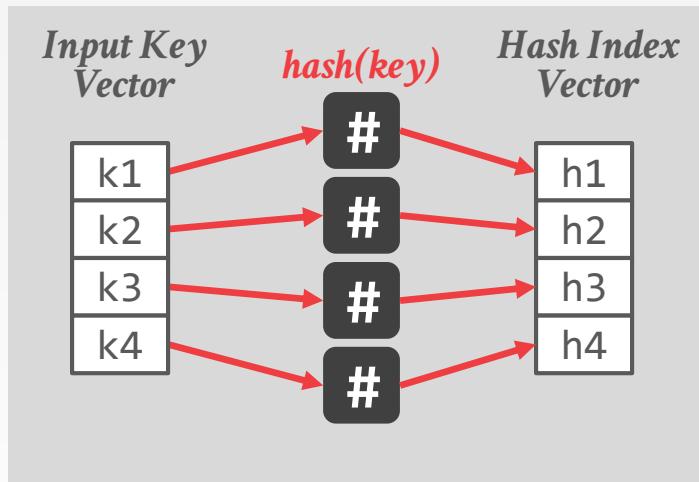


*Linear Probing Hash Table*

KEY	PAYLOAD
k99	
k1	
k6	
k4	
k5	
k88	

# HASH TABLES – PROBING

## *Vectorized (Vertical)*

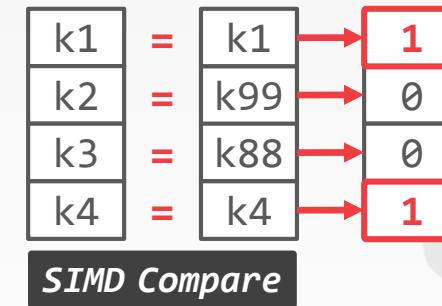
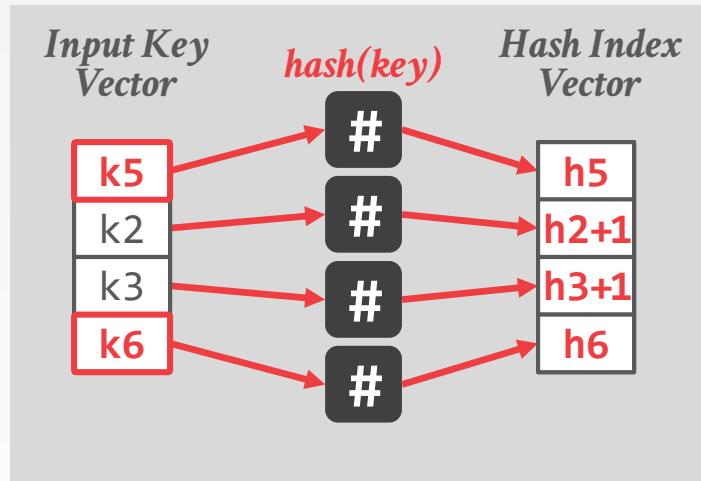


*Linear Probing Hash Table*

KEY	PAYLOAD
k99	
k1	
k6	
k4	
k5	
k88	

# HASH TABLES – PROBING

## *Vectorized (Vertical)*

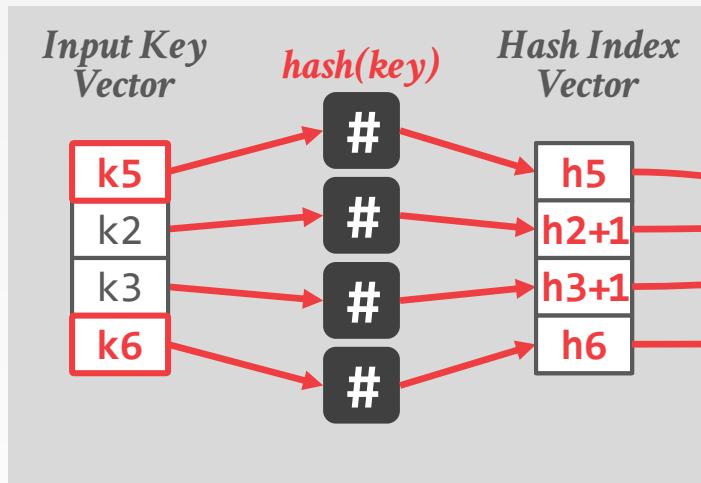


*Linear Probing Hash Table*

KEY	PAYOUT
k99	
k1	
k2	
k3	
k4	
k6	
k1	1
k99	0
k88	0
k4	1
k5	
k88	

# HASH TABLES – PROBING

## *Vectorized (Vertical)*



*Linear Probing Hash Table*

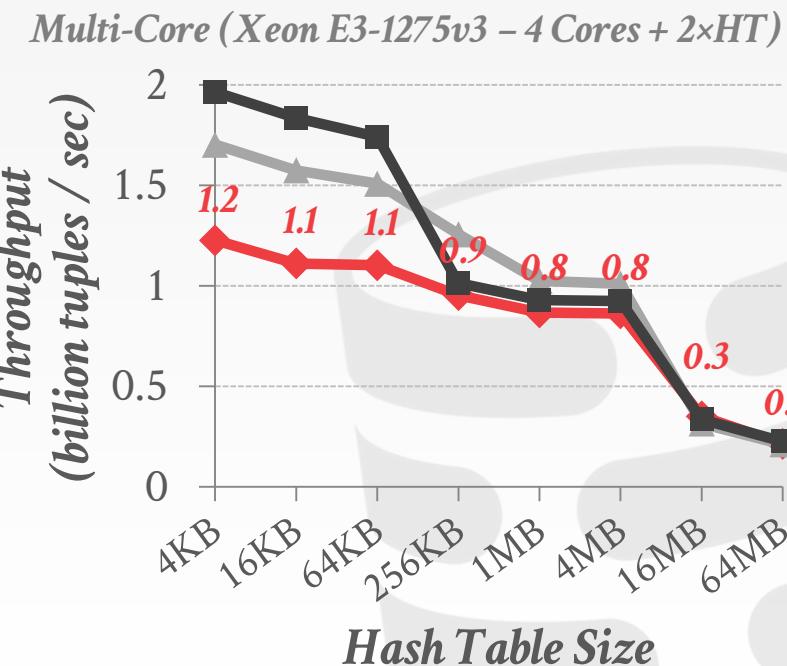
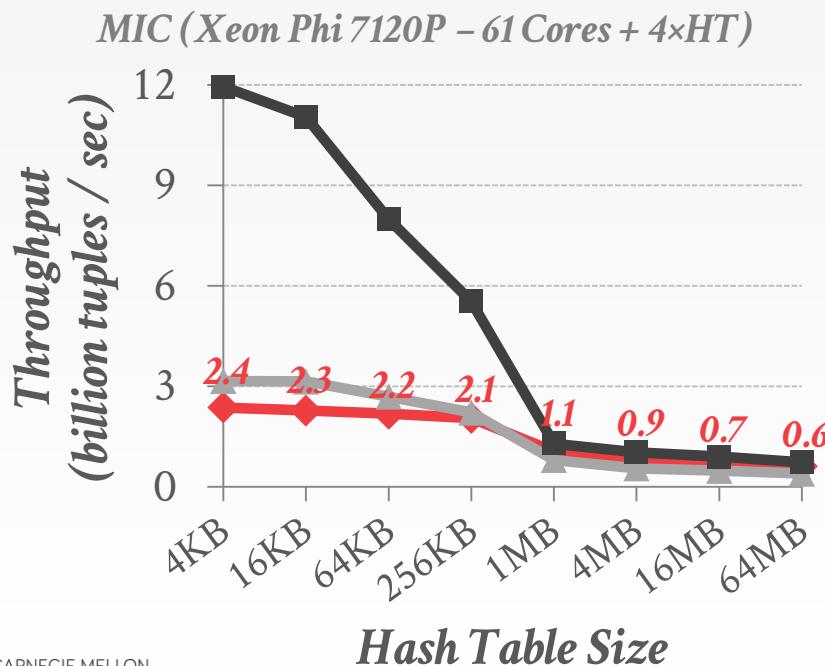
KEY	PAYOUT
k99	
k1	
k6	
k4	
k5	
k88	

# HASH TABLES – PROBING

◆ Scalar

▲ Vectorized (Horizontal)

■ Vectorized (Vertical)

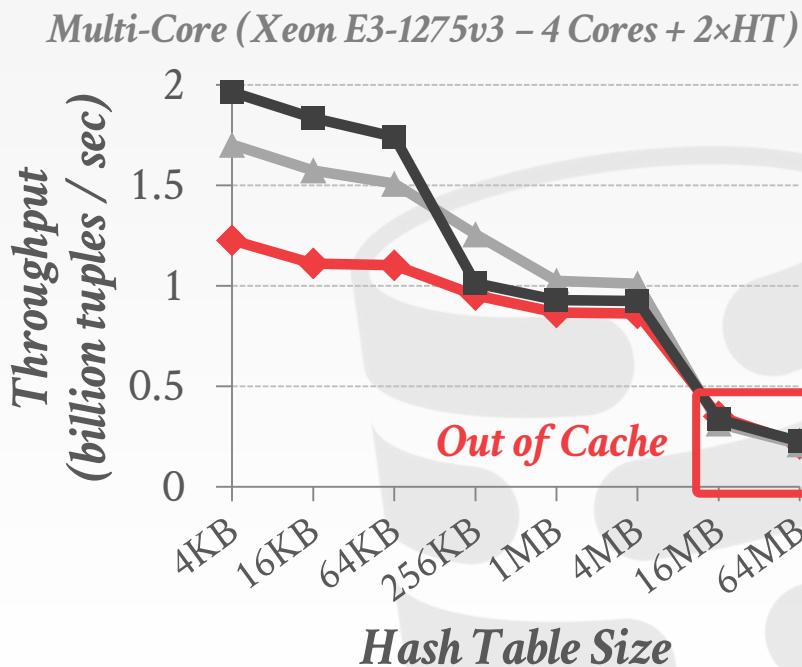
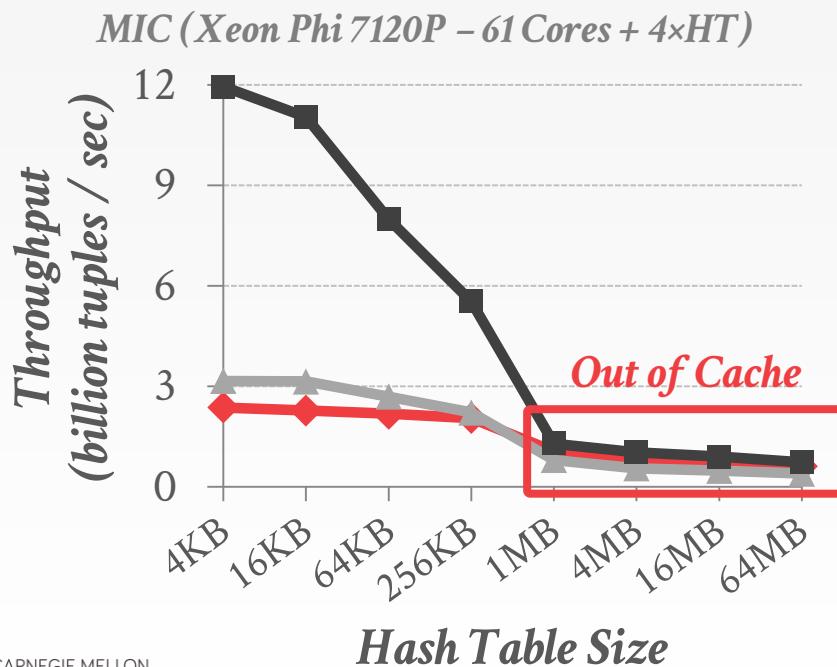


# HASH TABLES – PROBING

◆ Scalar

▲ Vectorized (Horizontal)

■ Vectorized (Vertical)



# PARTITIONING – HISTOGRAM

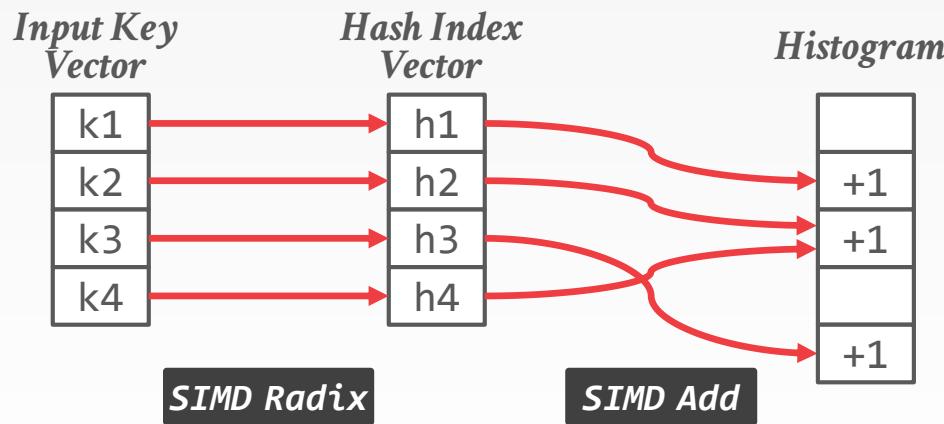
Use scatter and gathers to increment counts.

Replicate the histogram to handle collisions.



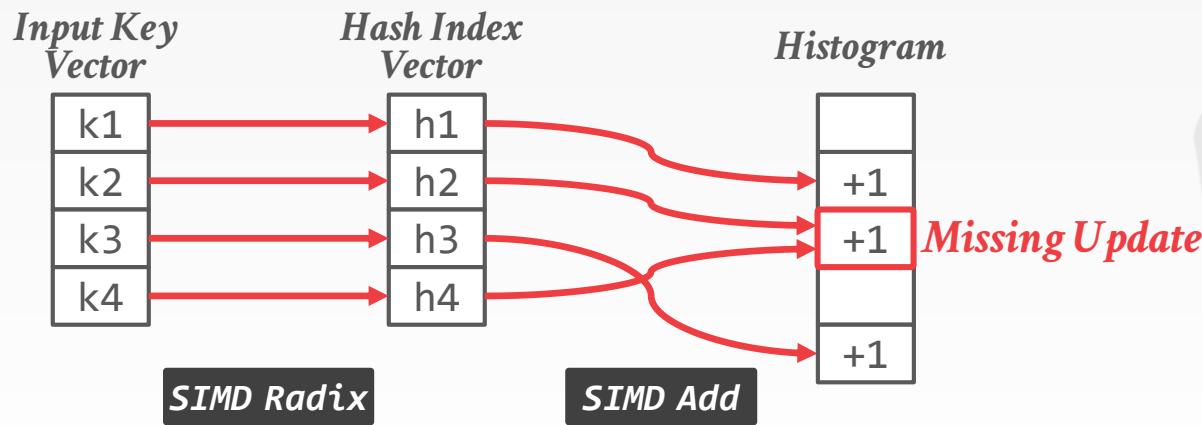
# PARTITIONING – HISTOGRAM

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



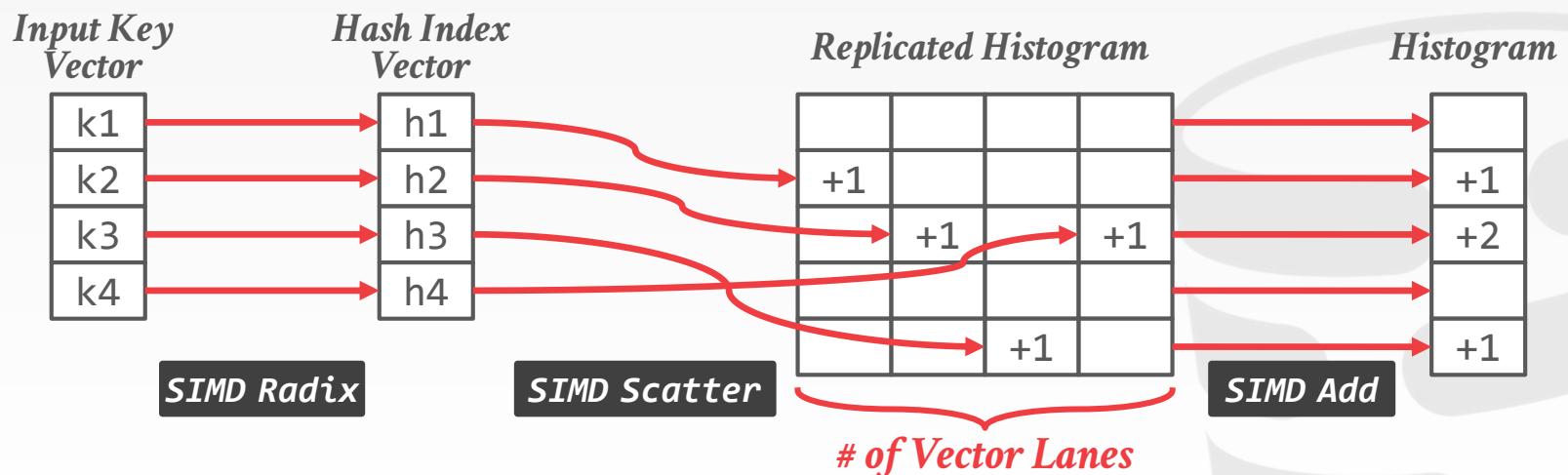
# PARTITIONING – HISTOGRAM

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



# PARTITIONING – HISTOGRAM

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



# JOINS

---

## No Partitioning

- Build one shared hash table using atomics
- Partially vectorized

## Min Partitioning

- Partition building table
- Build one hash table per thread
- Fully vectorized

## Max Partitioning

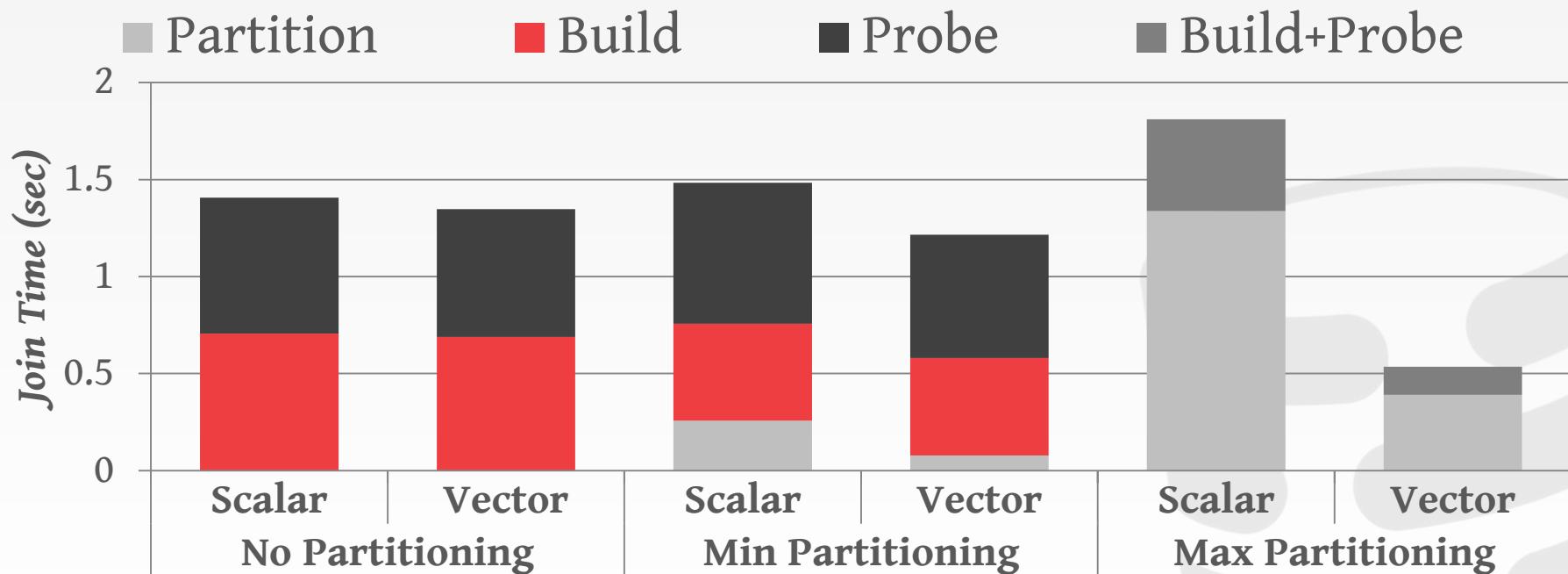
- Partition both tables repeatedly
- Build and probe cache-resident hash tables
- Fully vectorized



# JOINS

$200M \bowtie 200M$  tuples (32-bit keys & payloads)

Xeon Phi 7120P - 61 Cores + 4xHT



# PARTING THOUGHTS

---

Vectorization is essential for OLAP queries.

These algorithms don't work when the data exceeds your CPU cache.

We can combine all the intra-query parallelism optimizations we've talked about in a DBMS.

- Multiple threads processing the same query.
- Each thread can execute a compiled plan.
- The compiled plan can invoke vectorized operations.



# NEXT CLASS

---

Compilation vs. Vectorization

