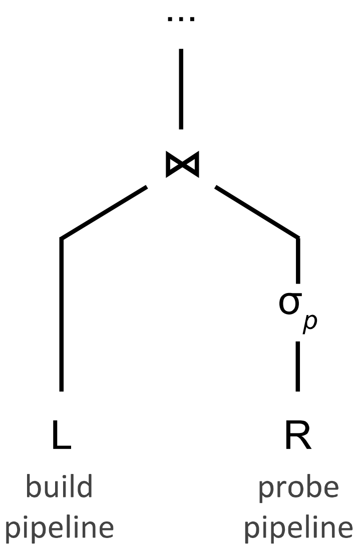
Make the Most out of Your SIMD Investments:
Counter Control Flow Divergence in Compiled Query Pipelines

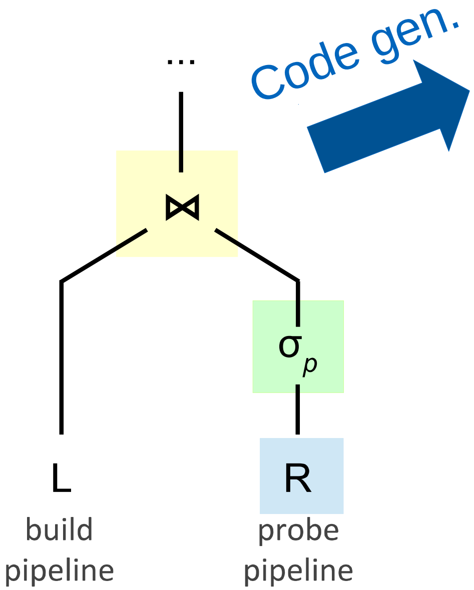
Harald Lang, Andreas Kipf, Linnea Passing,
Peter Boncz, Thomas Neumann, Alfons Kemper

DaMoN’18, June 11, 2018, Houston, TX, USA

Motivation

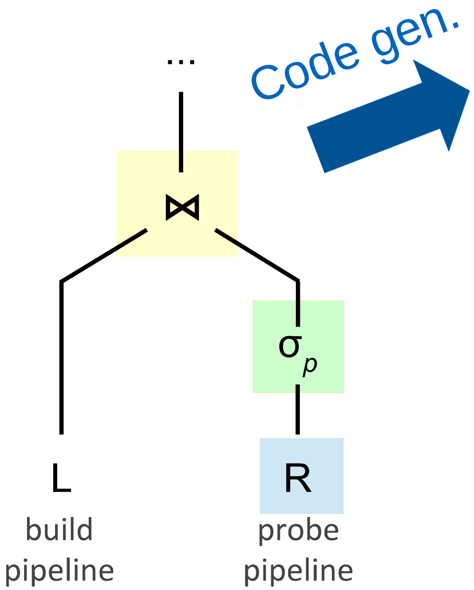


Motivation



```
// Tuple at a time
for each tuple in R
  if tuple satisfies p
    if join partner found
      … // subsequent operator
    end
  end
end
```

Motivation



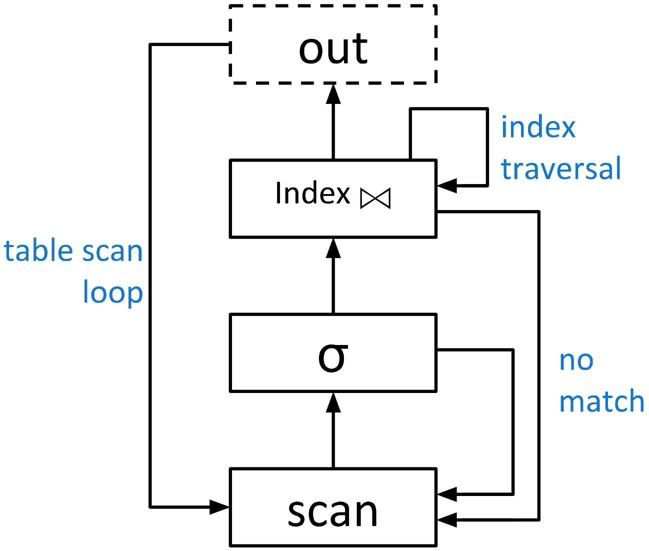
```
// Tuple at a time
for each tuple in R
  if tuple satisfies p
    if join partner found
      … // subsequent operator
    end
  end
end
```

Vectorization

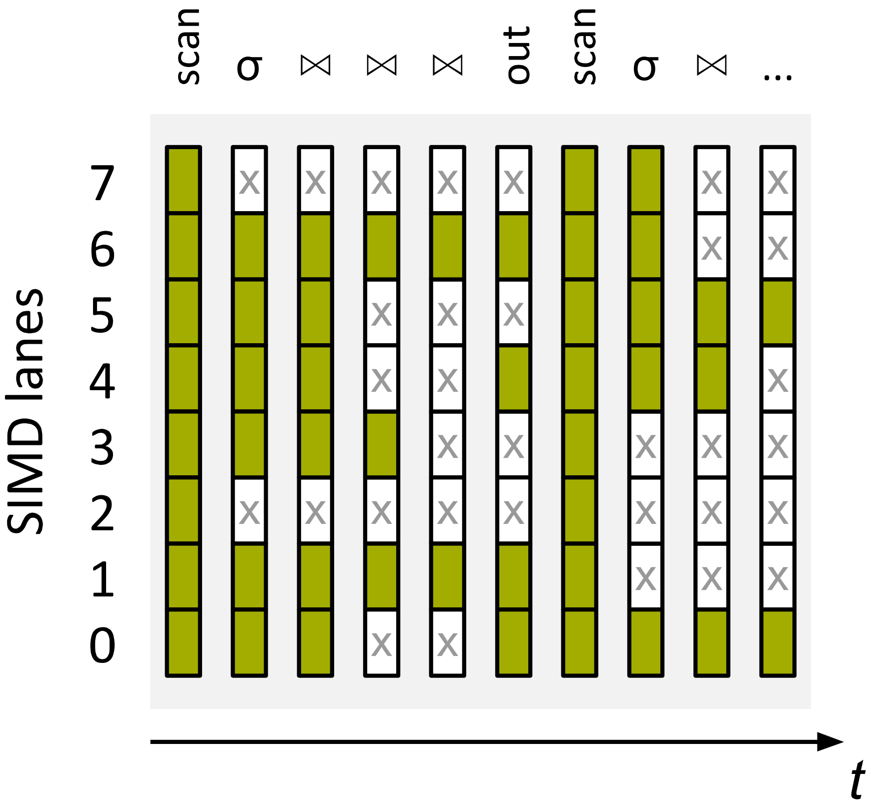
```
// (SIMD) Vector at a time
for each vector in R
  if at least one vector-element satisfies p
    if at least one vector-element
       has a join partner
      … // code of subsequent operator
    end
  end
end
```

Motivation (cont‘d)

Control-flow graph:



SIMD lane utilization:



	scan	σ	⋈	⋈	⋈	out	scan	σ	⋈	...
7	🟩	X	X	X	X	X	🟩	🟩	X	X
6	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	X	X
5	🟩	🟩	🟩	X	X	X	🟩	🟩	🟩	🟩
4	🟩	🟩	🟩	X	X	🟩	🟩	🟩	🟩	X
3	🟩	🟩	🟩	🟩	X	X	🟩	X	X	X
2	🟩	X	X	X	X	X	🟩	X	X	X
1	🟩	🟩	🟩	🟩	🟩	🟩	🟩	X	X	X
0	🟩	🟩	🟩	X	X	🟩	🟩	🟩	🟩	🟩

Contributions

1) **Algorithms** for AVX-512 SIMD to „**refill**“ the gaps.

2) **Strategies** for integration with compiled query pipelines.

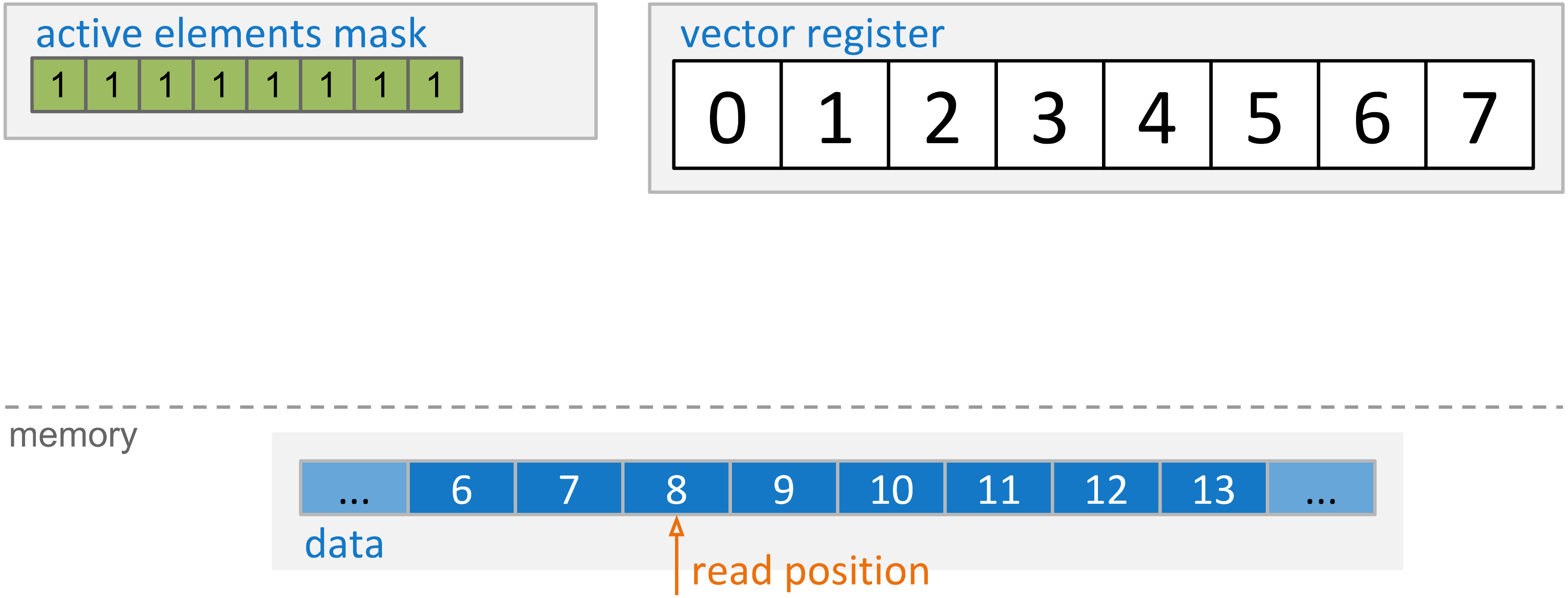
Algorithms to refill idle SIMD lanes

Refill Algorithms for idle SIMD Lanes

- Basic building blocks to counter underutilization
  - enabled by AVX-512 instruction set
  - possible with pre-AVX-512 architectures, but not efficient

- **Copy new elements to idle SIMD lanes**
  - at random positions
  - without altering/modifying active lanes.

Refill from Memory



active elements mask

1	1	1	1	1	1	1	1

vector register

0	1	2	3	4	5	6	7

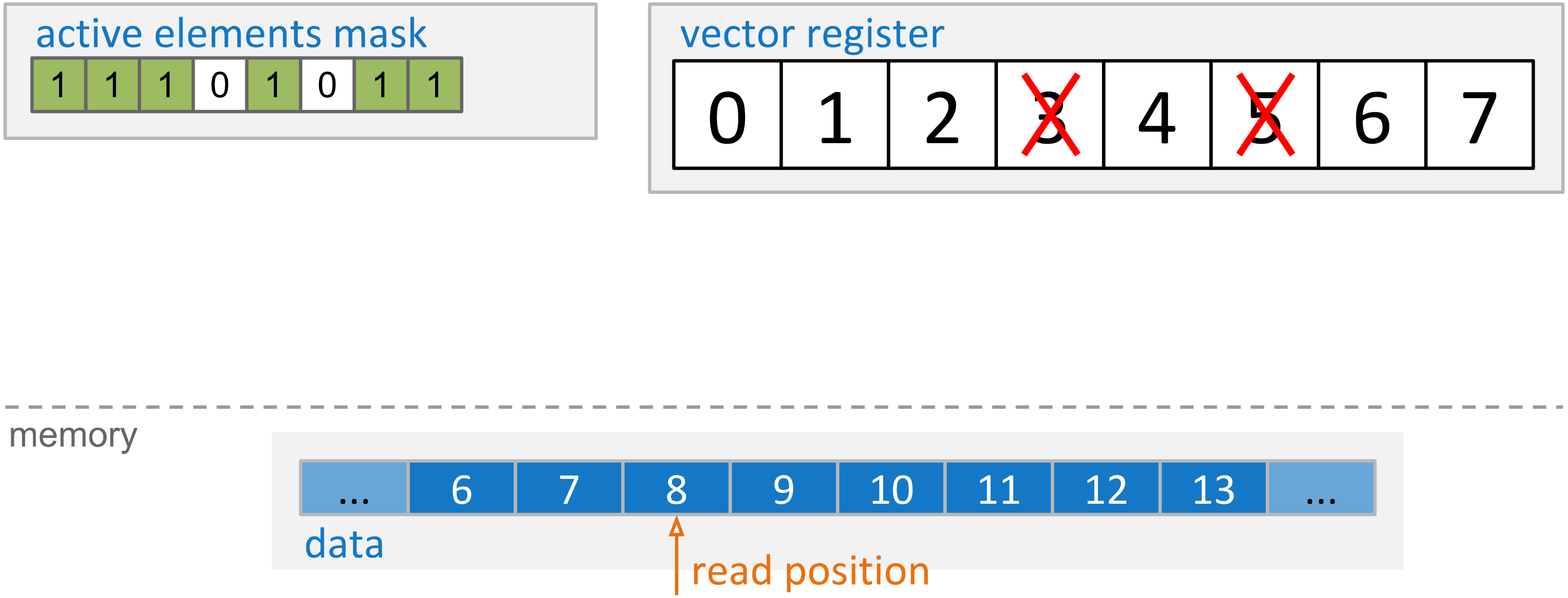
memory

…	6	7	8	9	10	11	12	13	…

data

read position

Refill from Memory



active elements mask

1	1	1	0	1	0	1	1

vector register

0	1	2	~~3~~	4	~~5~~	6	7

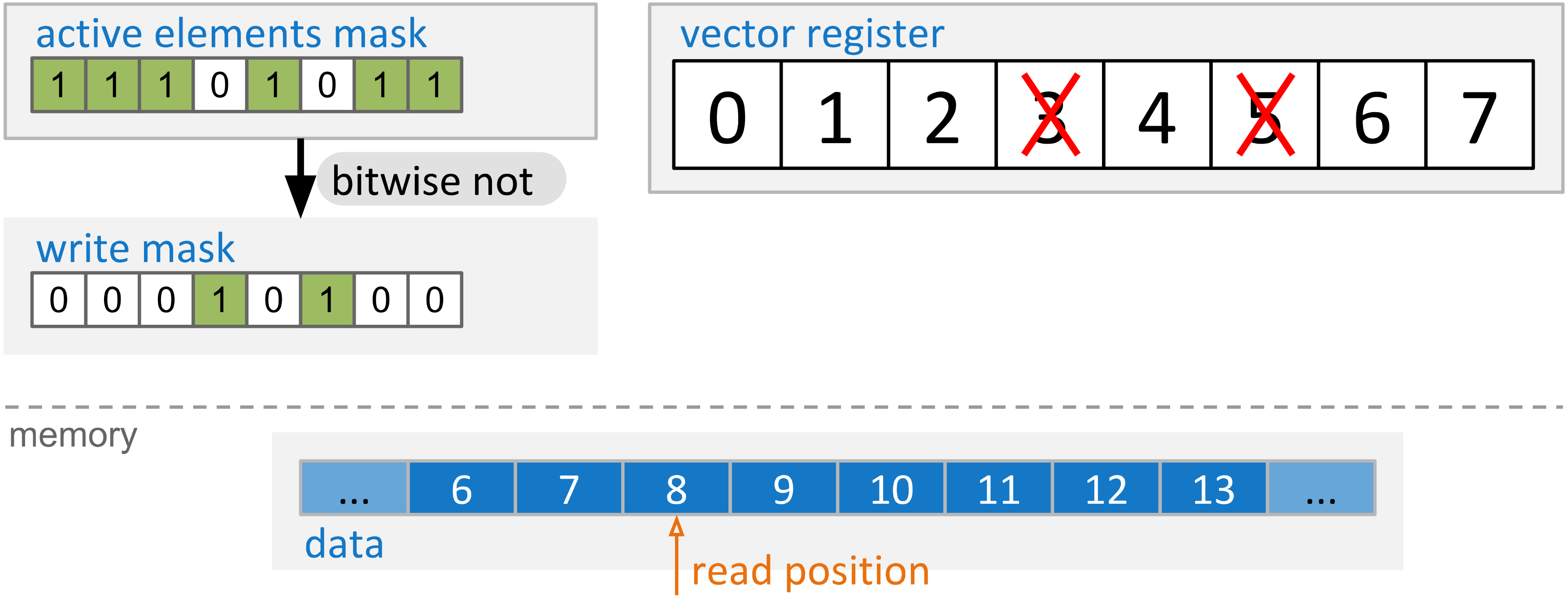
memory

…	6	7	8	9	10	11	12	13	…

data

read position

Refill from Memory



active elements mask

1 1 1 0 1 0 1 1

bitwise not

write mask

0 0 0 1 0 1 0 0

vector register

0 1 2 ~~3~~ 4 ~~5~~ 6 7

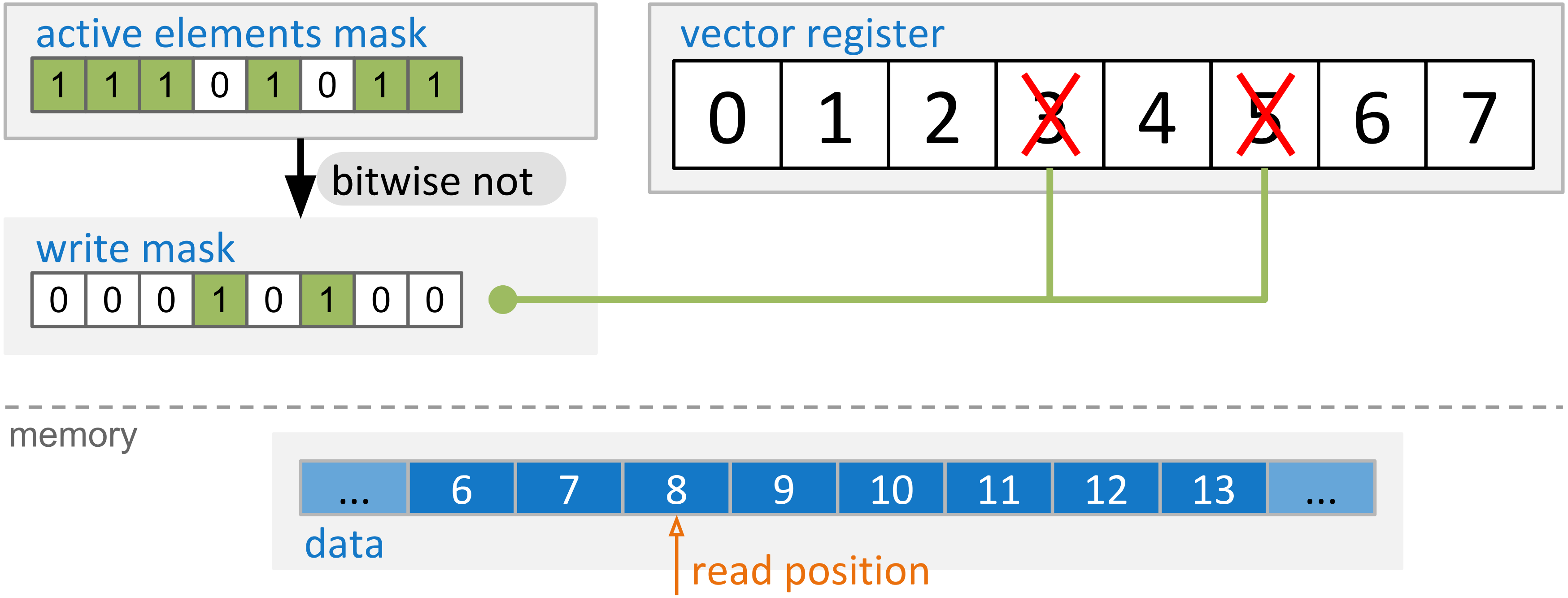
memory

... 6 7 8 9 10 11 12 13 ...

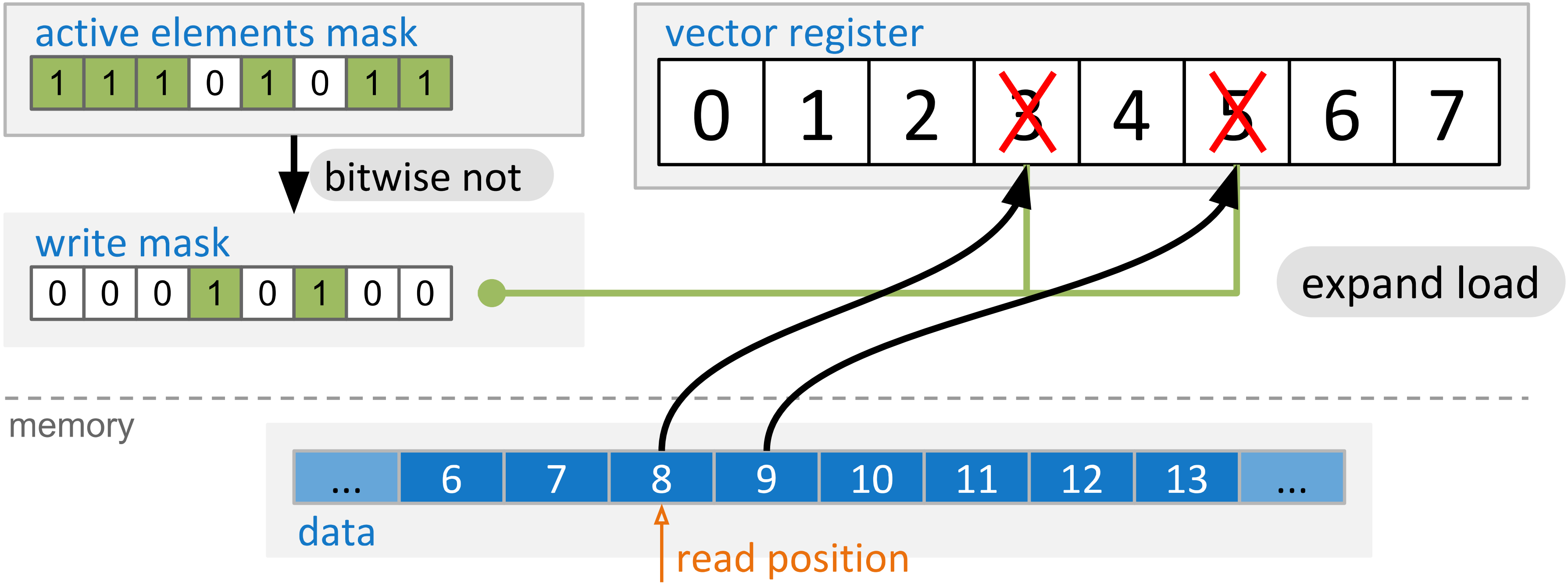
data

read position

Refill from Memory



Refill from Memory



active elements mask

1	1	1	0	1	0	1	1

bitwise not

write mask

0	0	0	1	0	1	0	0

vector register

0	1	2	~~3~~	4	~~5~~	6	7

expand load

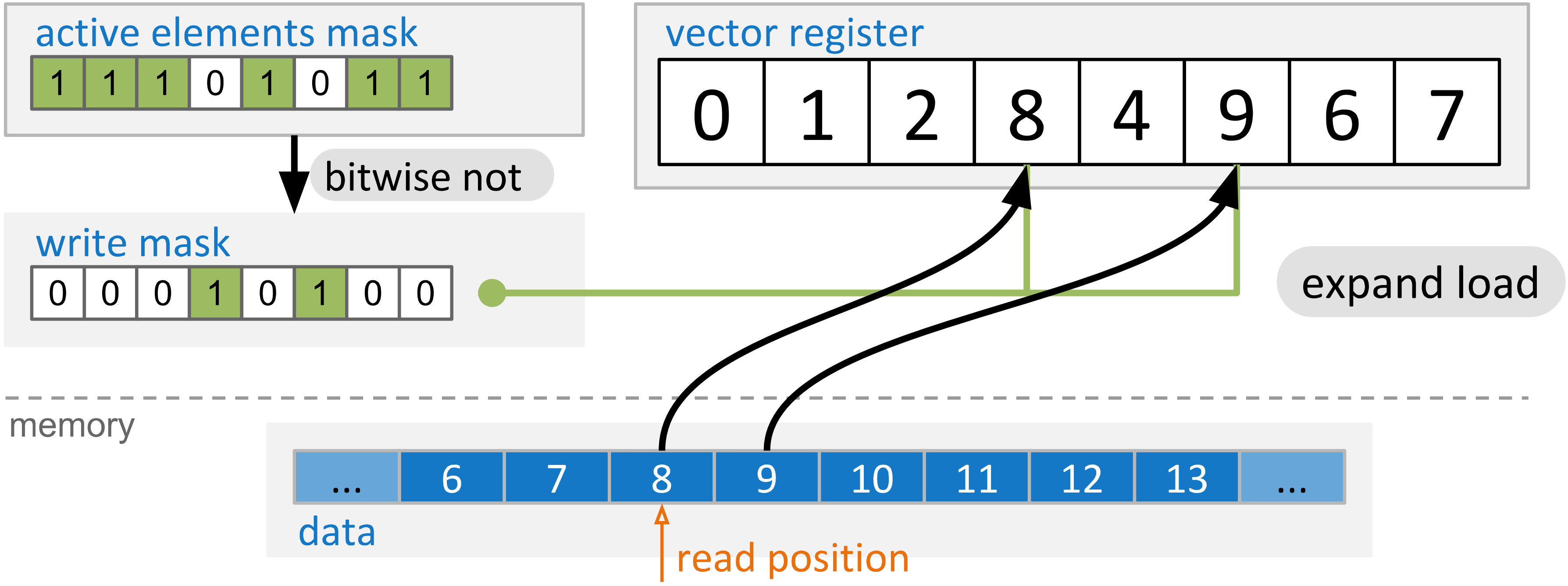
memory

...	6	7	8	9	10	11	12	13	...

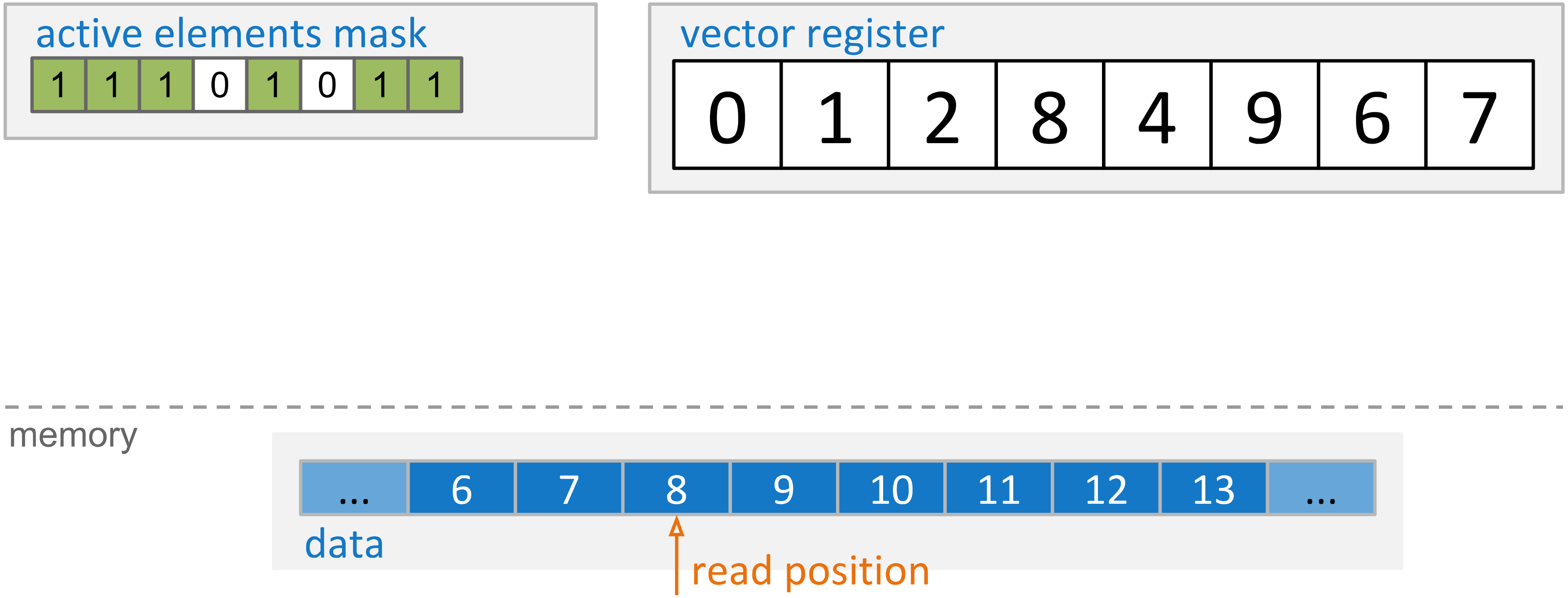
data

read position

Refill from Memory



Refill from Memory



active elements mask

1	1	1	0	1	0	1	1

vector register

0	1	2	8	4	9	6	7

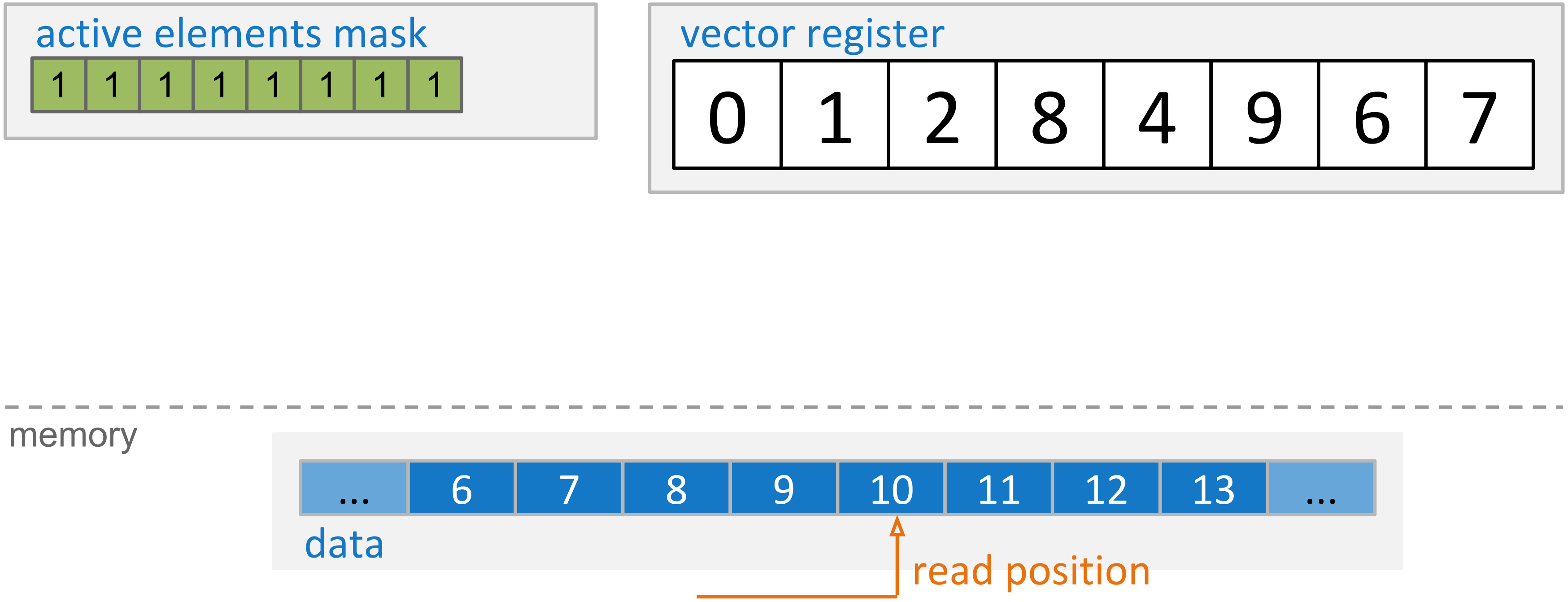
memory

…	6	7	8	9	10	11	12	13	…

data

read position

Refill from Memory



active elements mask

1 1 1 1 1 1 1 1

vector register

0 1 2 8 4 9 6 7

memory

… 6 7 8 9 10 11 12 13 …

data

read position

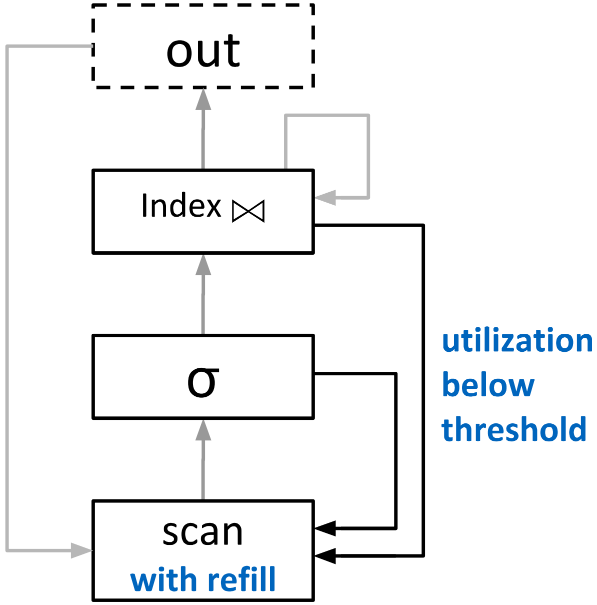
Refill Algorithms for idle SIMD Lanes

- Many different flavors, e.g.
  - copy from **memory to vector registers** (as shown)
  - copy **between vector registers** (more involved)

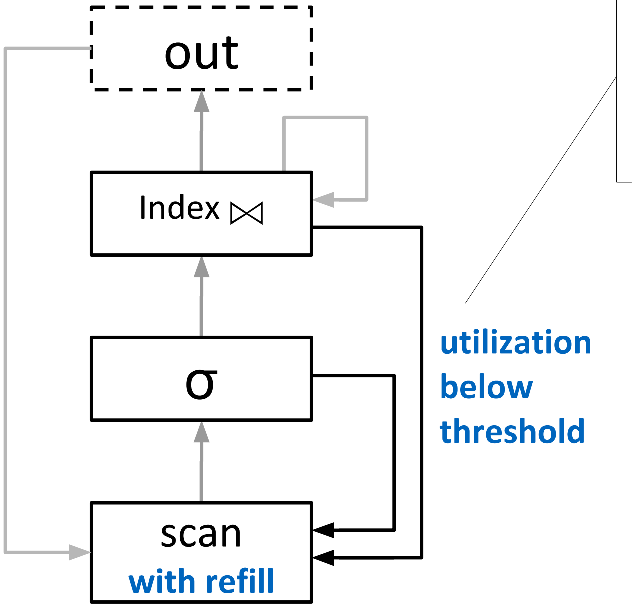
- **Implementations details**
  - in the paper
  - on GitHub: https://github.com/harald-lang/simd_divergence
  - in today‘s **poster session**: 3:30 pm – 4 pm

Strategies for integrating refills with compiled query pipelines.

1st Strategy: Refill from pipeline source

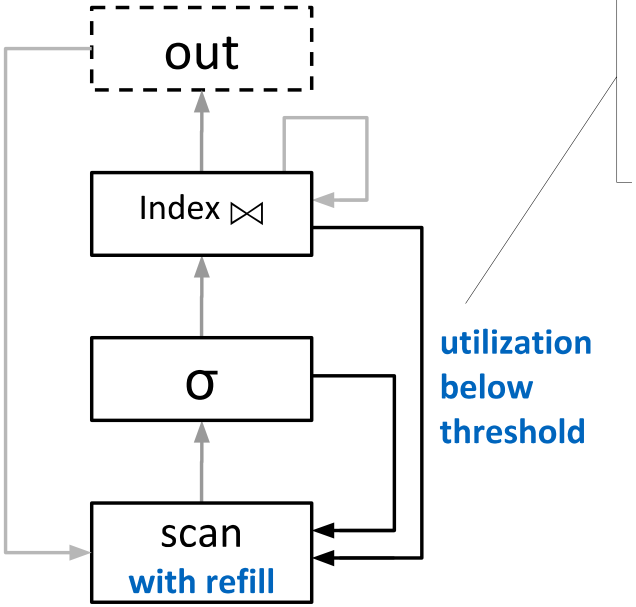


1st Strategy: Refill from pipeline source



If lane utilization falls below threshold, the **control flow is returned to the pipeline source** (e.g., table scan)

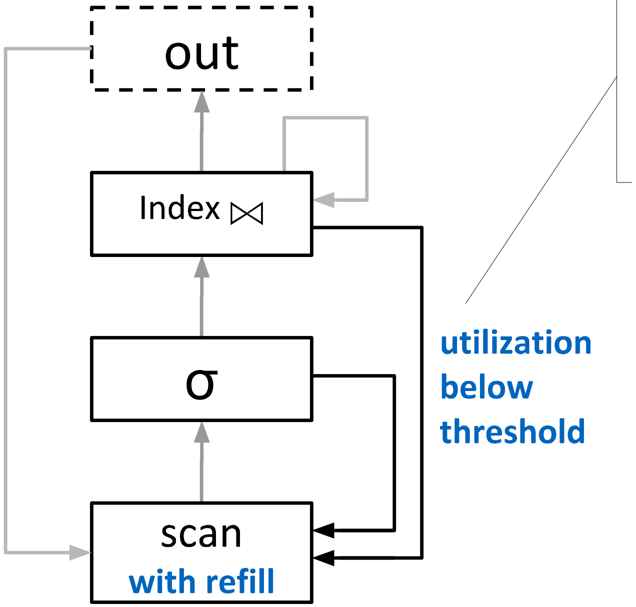
1st Strategy: Refill from pipeline source



If lane utilization falls below threshold, the **control flow is returned to the pipeline source** (e.g., table scan)

Refill idle SIMD lanes
→ typically from memory (as shown before)

1st Strategy: Refill from pipeline source

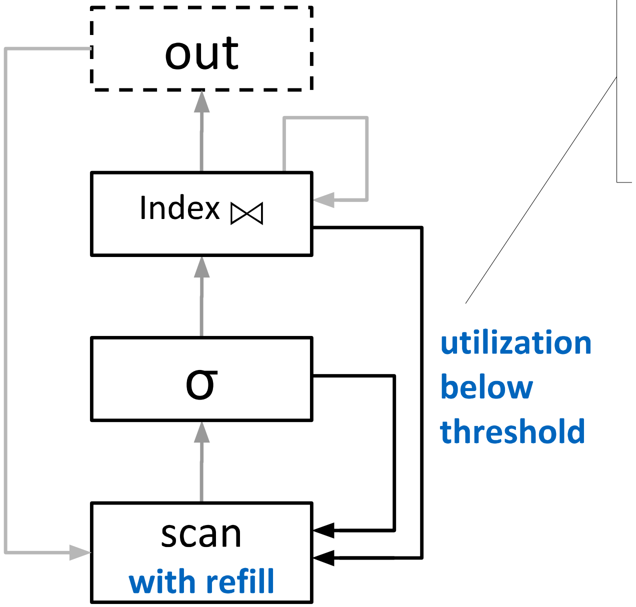


If lane utilization falls below threshold, the **control flow is returned to the pipeline source** (e.g., table scan)

Refill idle SIMD lanes
→ typically from memory (as shown before)

➡ Active elements remain in vector registers.

1st Strategy: Refill from pipeline source



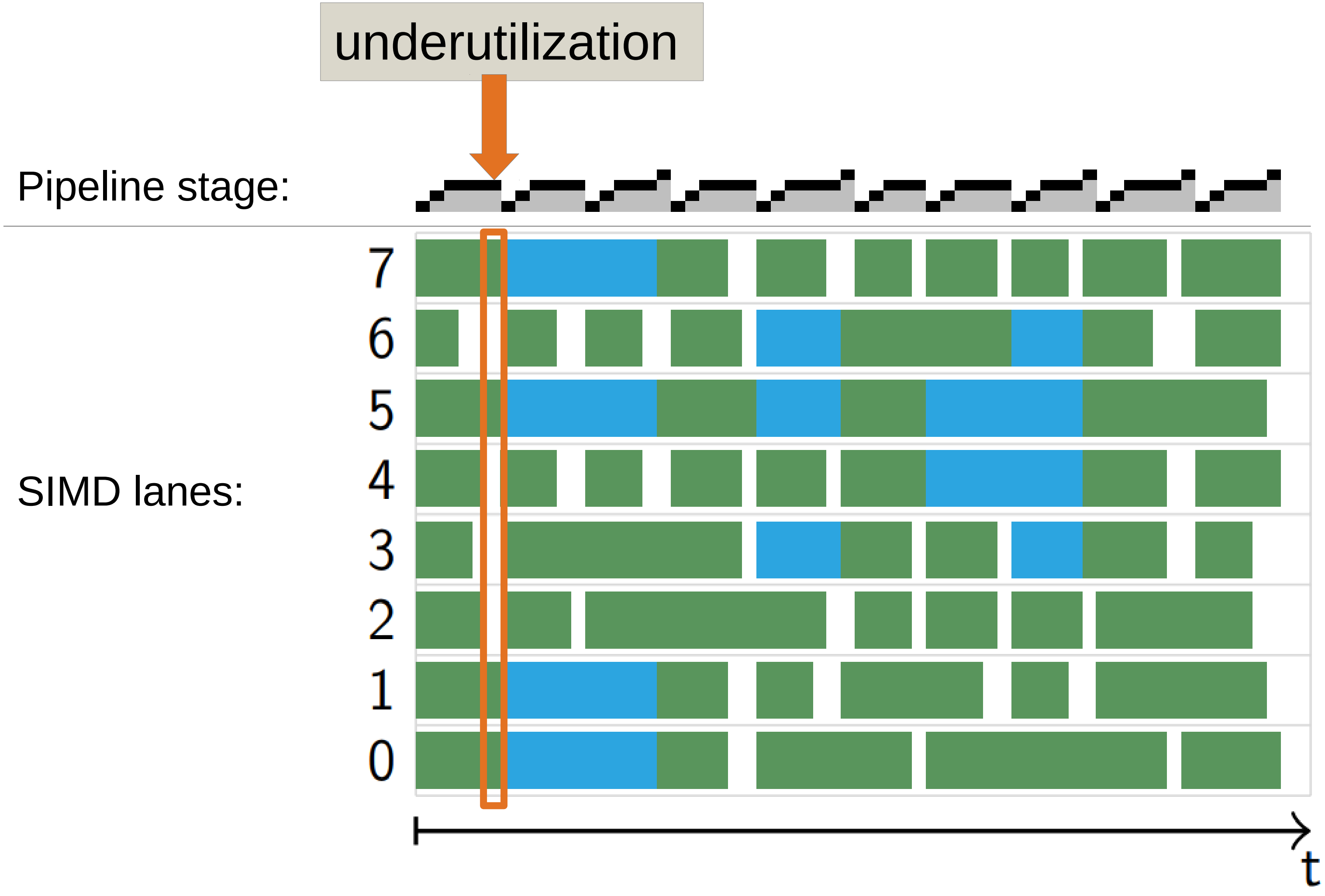
If lane utilization falls below threshold, the **control flow is returned to the pipeline source** (e.g., table scan)

Refill idle SIMD lanes
→ typically from memory (as shown before)

⇨ Active elements remain in vector registers.

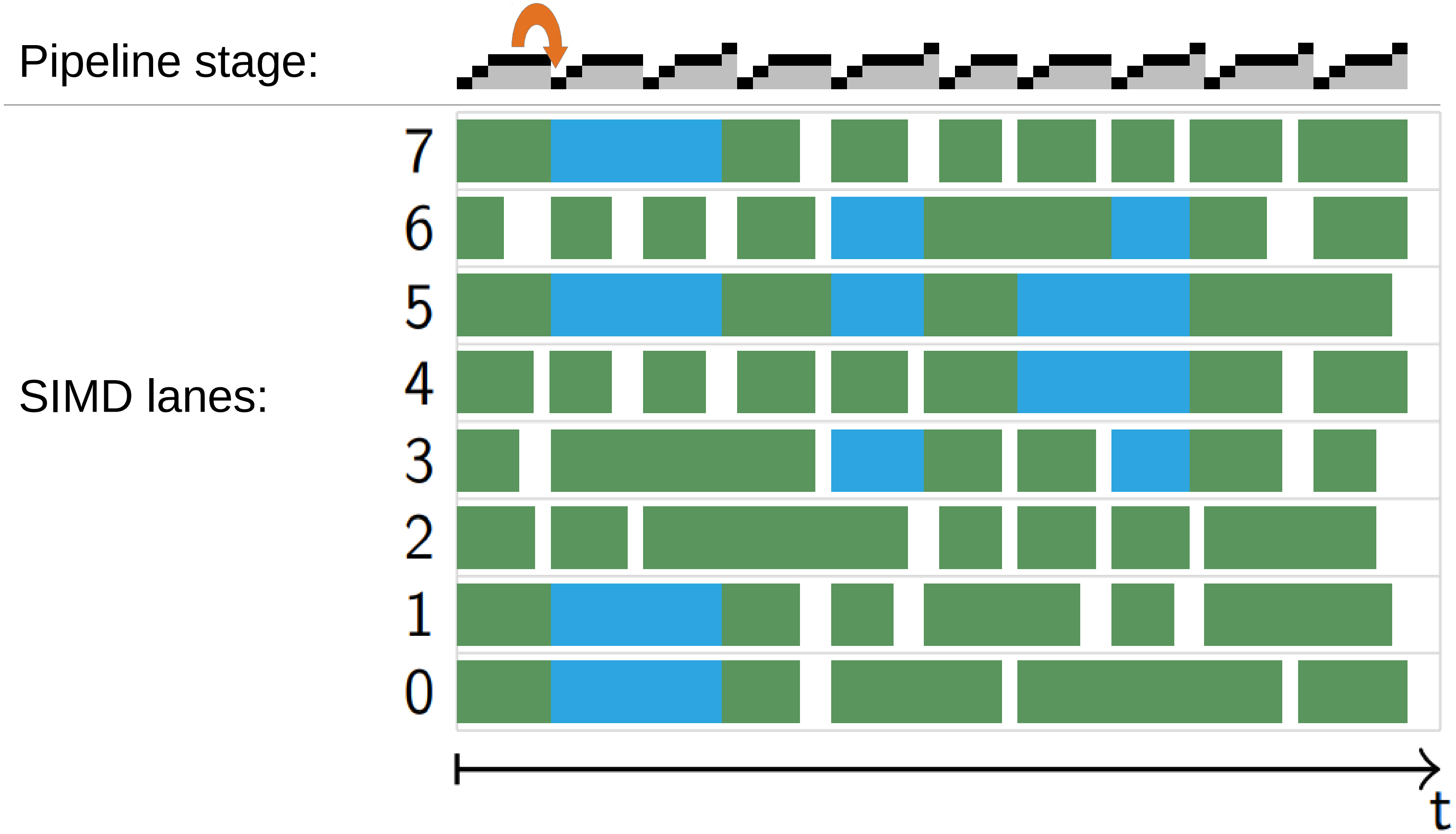
⇨ Lanes must be **protected** from being modified.

1st Strategy: Refill from pipeline source

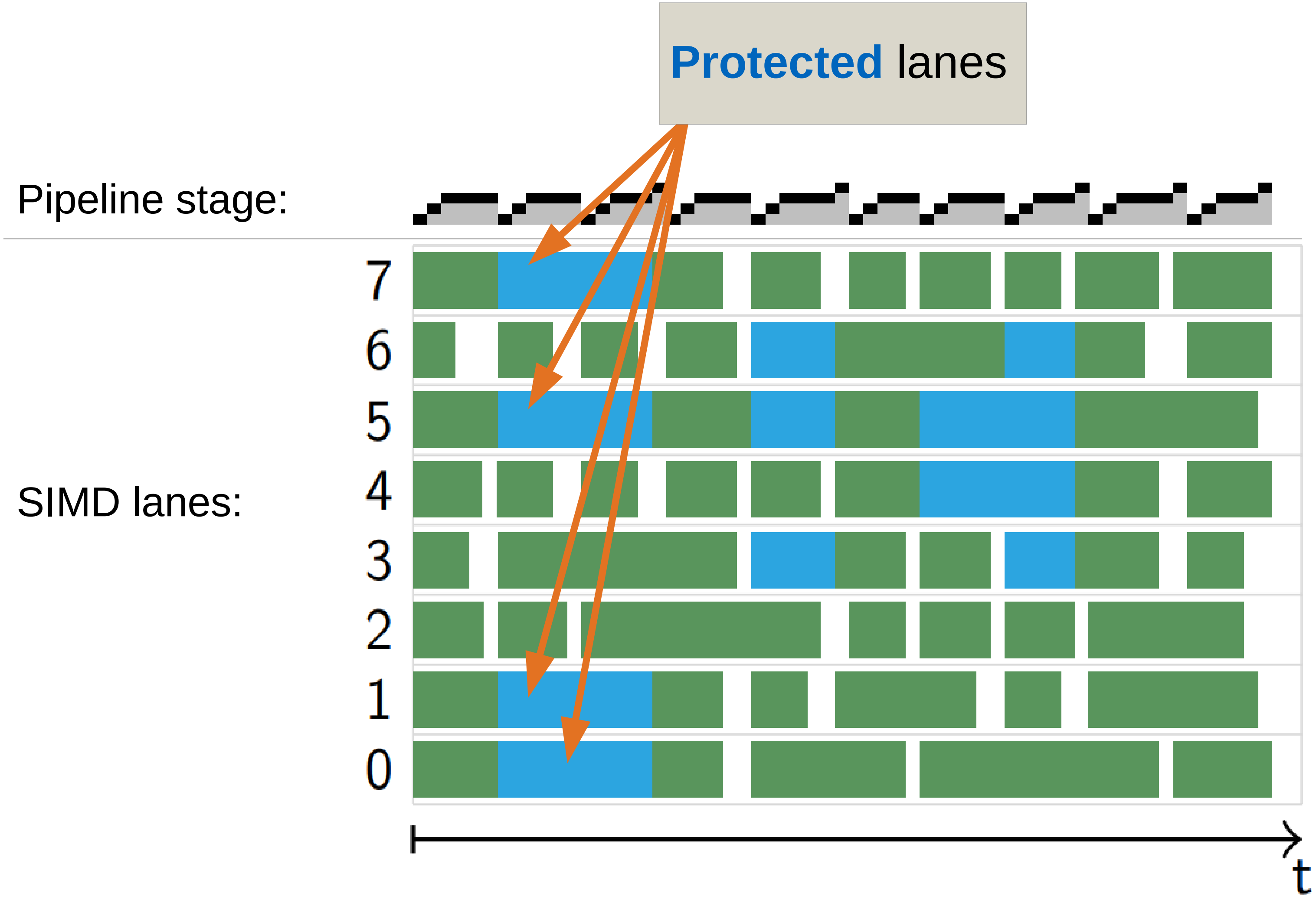


1st Strategy: Refill from pipeline source

Control flow returns to table scan



1st Strategy: Refill from pipeline source

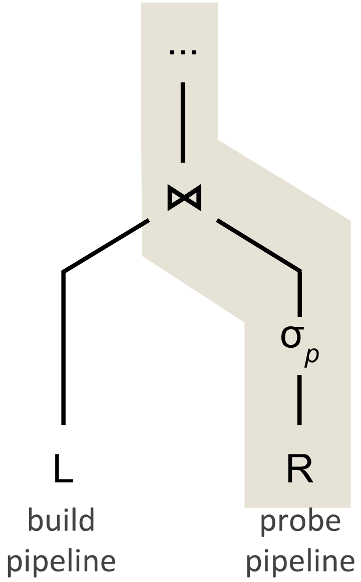


1st Strategy: Refill from pipeline source

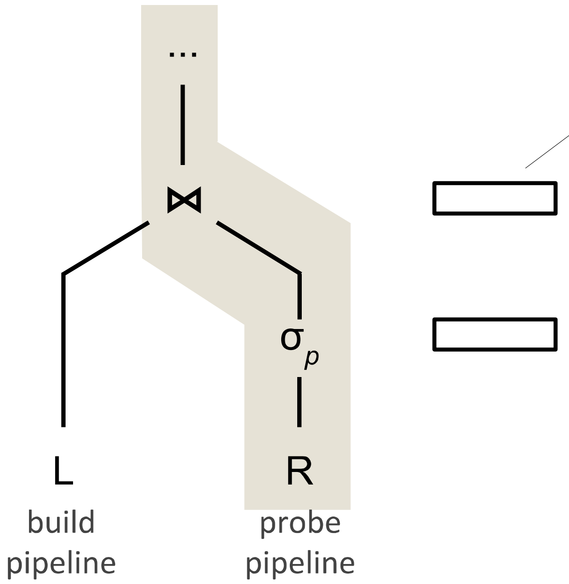
- **Lane protection** requires just a bit of bookkeeping
- Drawback: Inherently **causes underutilization** between the source and the operator that bailed out.
  → more costly, the longer the pipeline is
- Should only be used „close“ to the pipeline source.

Partial Consume Strategy

2nd Strategy: Refill from buffer

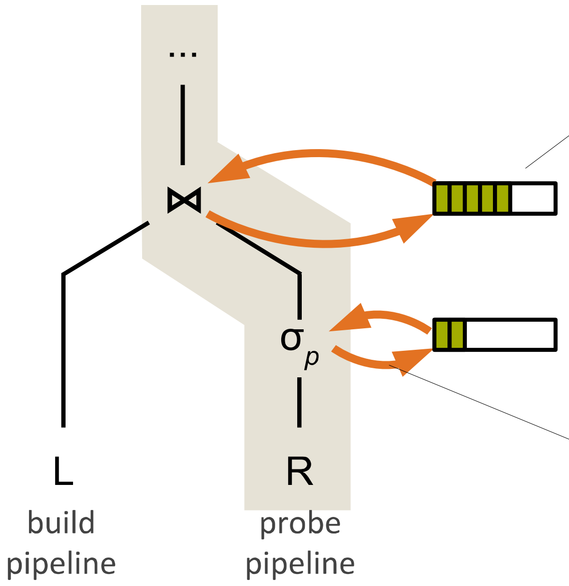


2nd Strategy: Refill from buffer



Operators allocate additional **vector register(s)** which are used as tiny **buffers.**

2nd Strategy: Refill from buffer

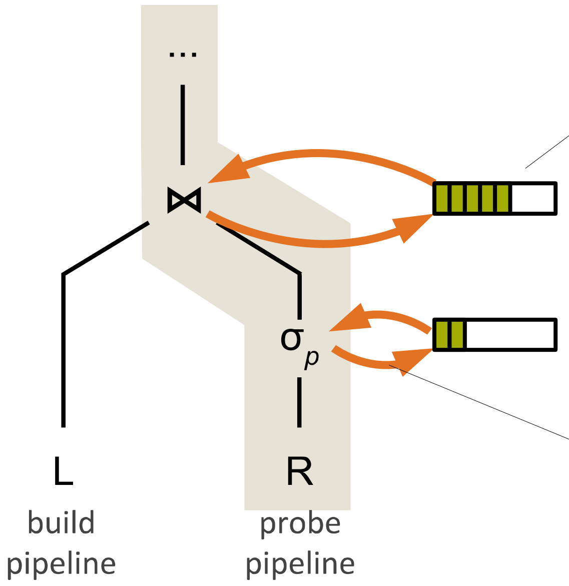


Operators allocate additional **vector register(s)** which are used as tiny **buffers.**

Buffers are used to:

- refill empty lanes, if enough elements are available
- buffer active elements before bailing out due to underutilization

2nd Strategy: Refill from buffer



Operators allocate additional **vector register(s)** which are used as tiny **buffers.**

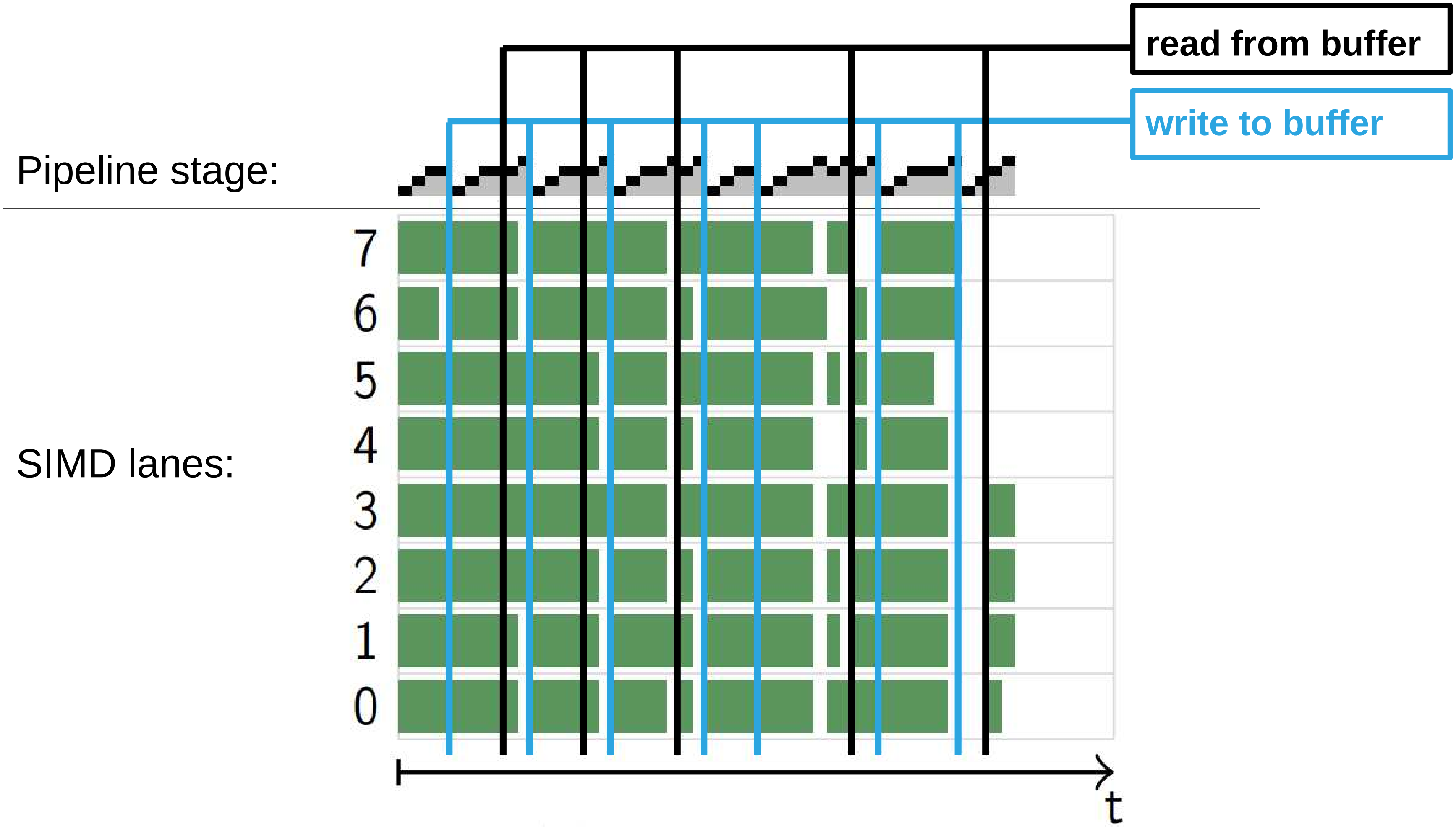
Buffers are used to:

- refill empty lanes, if enough elements are available
- buffer active elements before bailing out due to underutilization

⇨ All SIMD **lanes are empty** when the control flow returns

Consume Everything Strategy

2nd Strategy: Refill from buffer

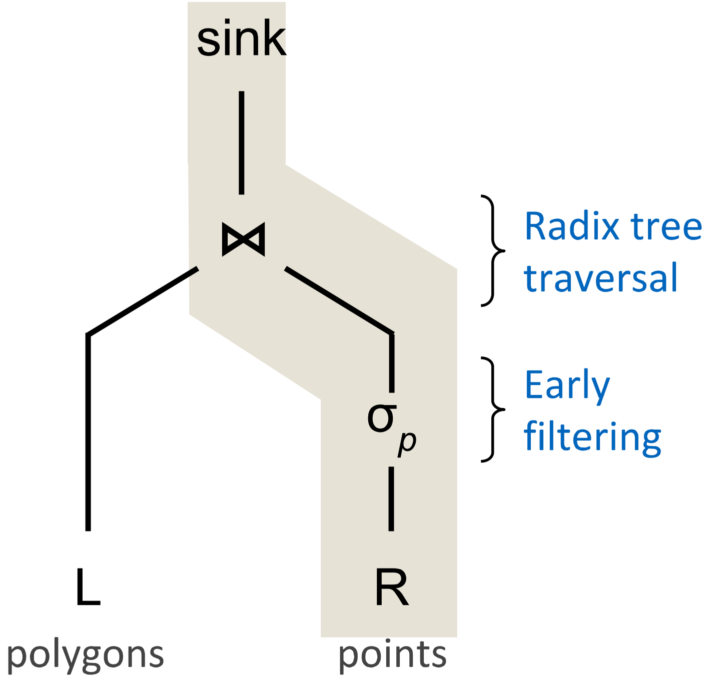


Mixed Strategy

- Both strategies can be applied within the same pipeline (**Mixed** strategy)
  - **Partial Consume** with lane protection should be used in operators close to the pipeline source,
  - **Consume Everything** with buffering, otherwise.

Evaluation – (approx.) point-polygon join

- **Polygons**: NYC boroughs, neighborhoods, census blocks, and manhattan (combines census blocks and neighborhood polys)
- **Points**: Random (uniformly distributed within the bounding box)
- **Hardware**:
  - Intel Knights Landing (Phi 7210)
  - Intel Skylake-X (i9-7900X)



Evaluation – (cont‘d)

Workload: Manhattan polygons, 15 meter precision

System	**Performance Baseline** (AVX-512)
Knights Landing Phi 7210	**3559 Mtps**
Skylake-X i9-7900X	**910 Mtps**

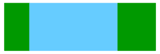
Evaluation – (cont‘d)

Workload: Manhattan polygons, 15 meter precision

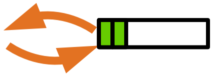
System	**Performance Baseline** (AVX-512)	**Improvement**
Knights Landing Phi 7210	**3559 Mtps**	**+ 20 %**
Skylake-X i9-7900X	**910 Mtps**	**+ 35 %**

Conclusions

- Control flow divergence wastes precious CPU resources
- Refilling empty SIMD lanes is important (and efficient since AVX-512)
- Integrates well with compiled query pipelines



use **Partial Consume with lane protection** in the very first operators (close to the pipeline source)



and apply **Consume Everything with buffering** otherwise. In particular, when traversing irregular pointer based data structures.

Thank You!

Q & A