Paper reading group

Vanilla attention basics and llama KV cache design

Joel Fernandes

Paper reading group

Vanilla attention basics and llama KV cache design

Joel Fernandes

one pool of cells, many sequences



Paper reading group

Vanilla attention basics and llama KV cache design

Joel Fernandes

one pool of cells, many sequences



Attention & KV cache
slides 1–6

Paper reading group

Vanilla attention basics and llama KV cache design

Joel Fernandes

one pool of cells, many sequences



Attention & KV cache
slides 1–6

Unified vs split
7–8

Mask & kv_rows
9–15

Paper reading group

Vanilla attention basics and llama KV cache design

Joel Fernandes

one pool of cells, many sequences



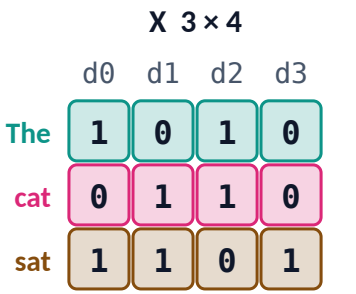
Attention & KV cache
slides 1–6

Unified vs split
7–8

Mask & kv_rows
9–15

Three projections: q, k and v

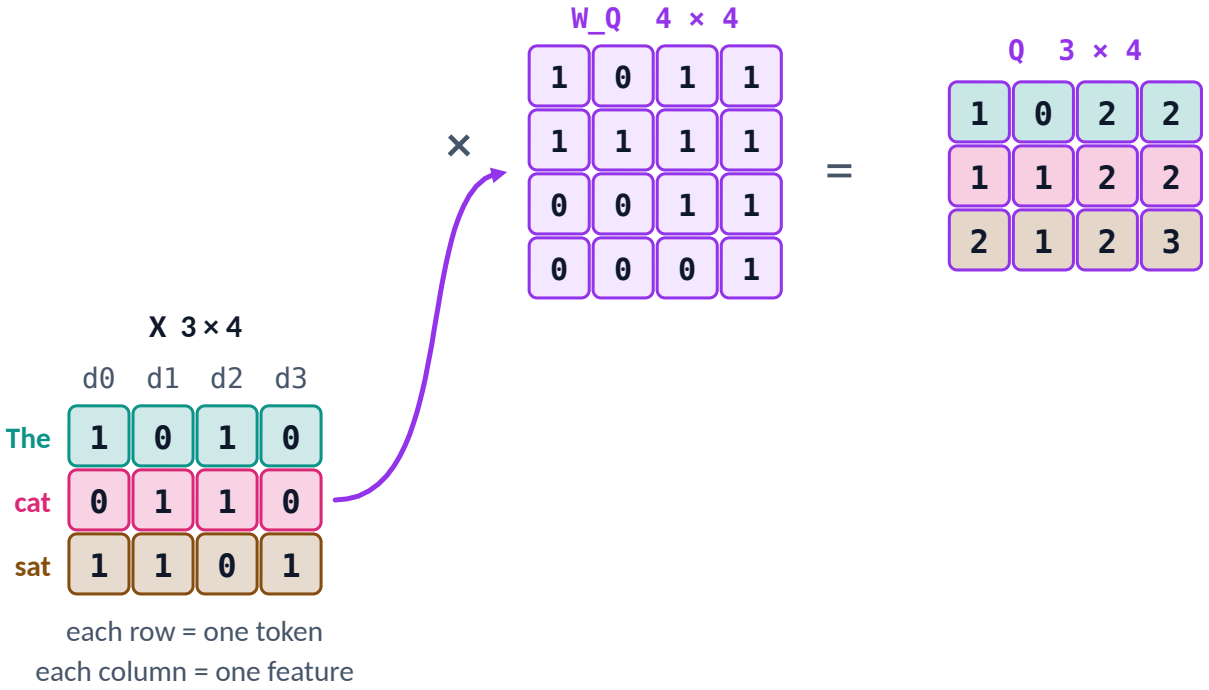
X: each token’s vector, times three learned 4 × 4 matrices.



each row = one token
each column = one feature

Three projections: q, k and v

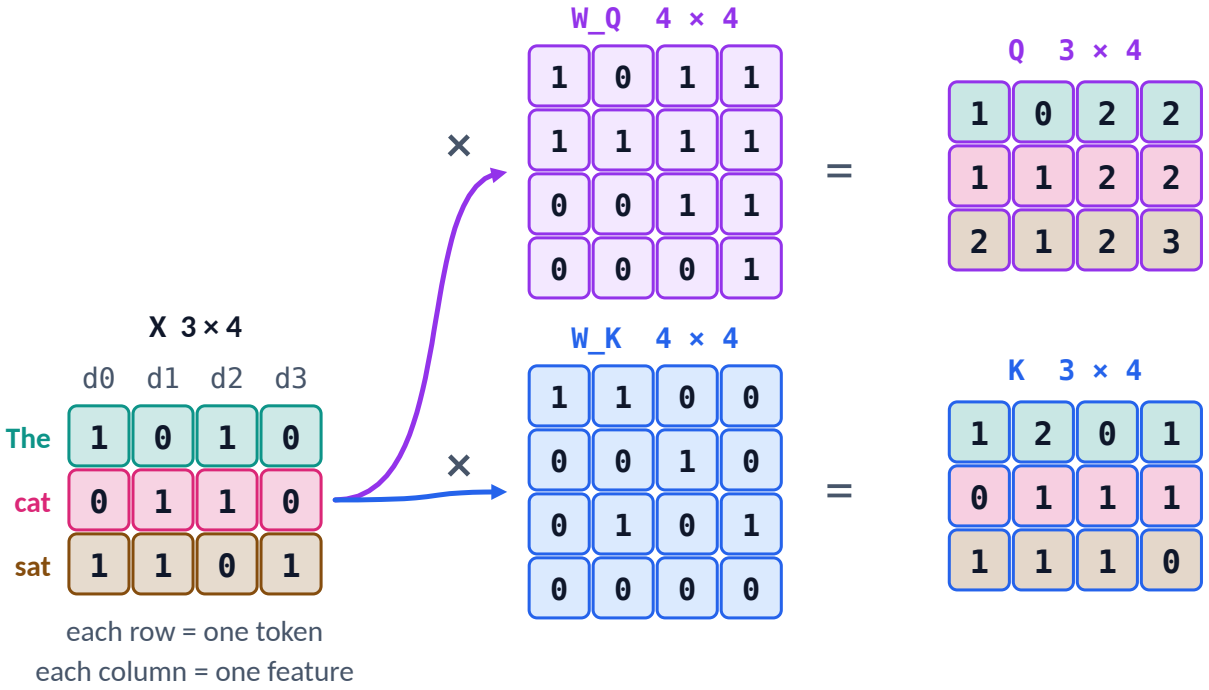
X: each token's vector, times three learned 4 × 4 matrices.



q: what I look for

Three projections: q, k and v

X: each token's vector, times three learned 4 × 4 matrices.

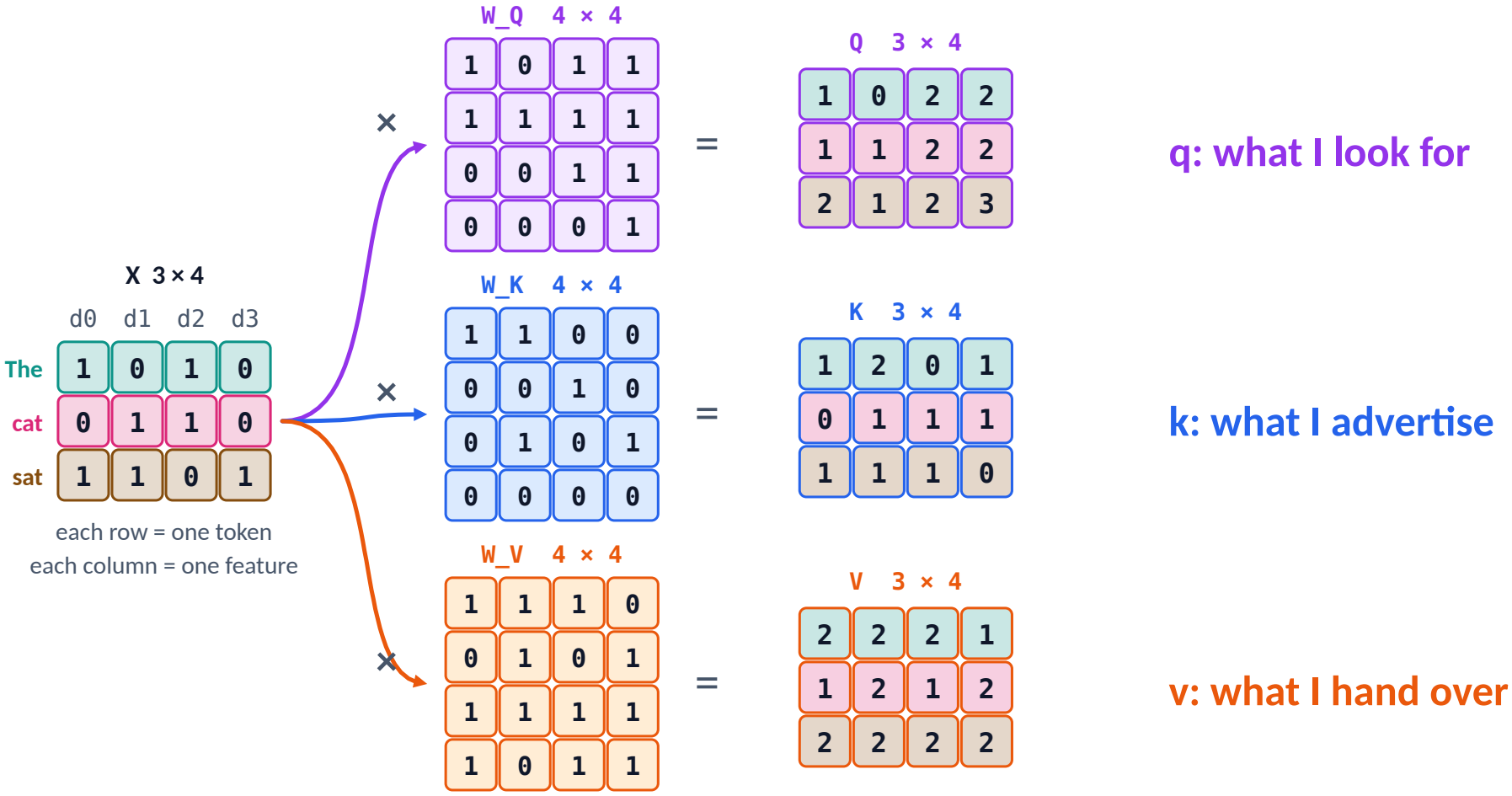


q: what I look for

k: what I advertise

Three projections: q, k and v

X: each token's vector, times three learned 4 × 4 matrices.



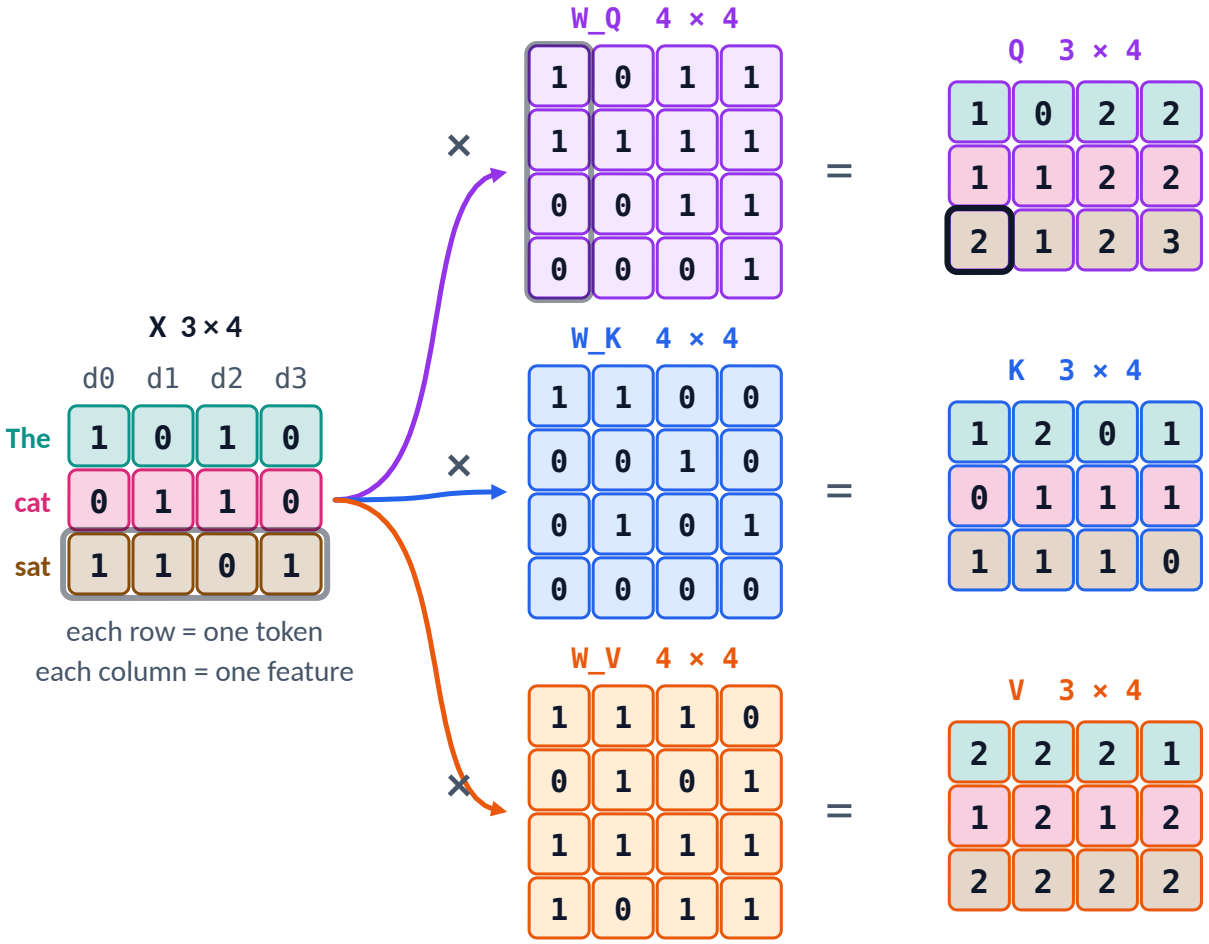
q: what I look for

k: what I advertise

v: what I hand over

Three projections: q, k and v

X: each token's vector, times three learned 4 × 4 matrices.



q: what I look for

k: what I advertise

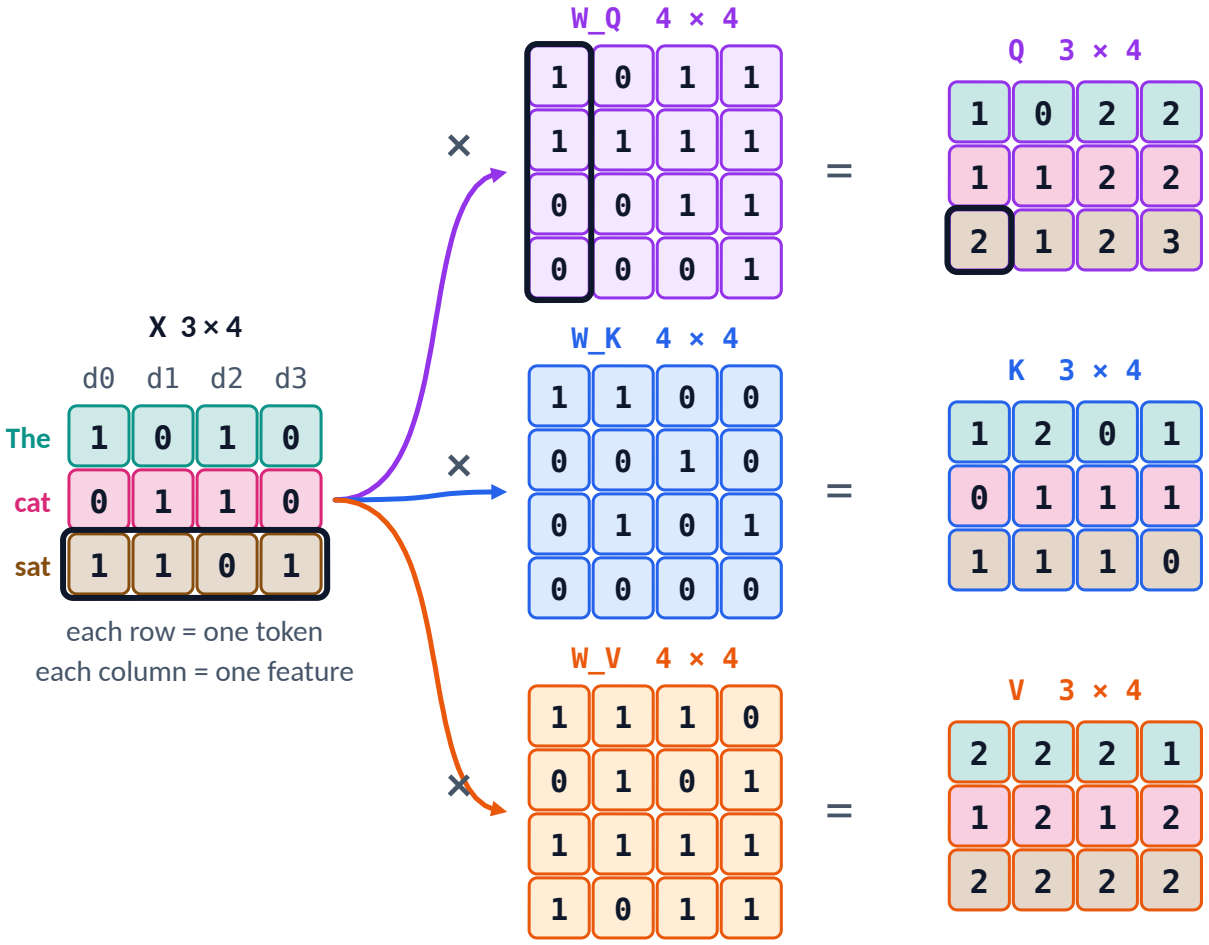
v: what I hand over

$$x_{sat} \cdot W_Q[:, 0] = 1 \cdot 1 + 1 \cdot 1 + 0 \cdot 0 + 1 \cdot 0 = 2$$

row of X · column of W = one number in Q

Three projections: q, k and v

X: each token's vector, times three learned 4 × 4 matrices.



q: what I look for

k: what I advertise

v: what I hand over

$$x_{sat} \cdot W_Q[:, 0] = 1\cdot1 + 1\cdot1 + 0\cdot0 + 1\cdot0 = 2$$

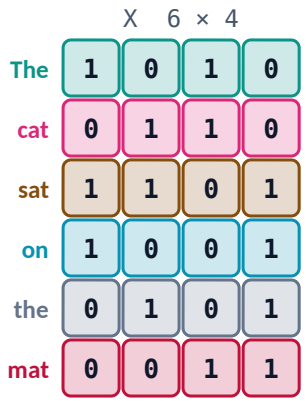
row of X · column of W = one number in Q

real model: 5120 numbers per token, many heads, 4 carry k and v

One forward pass

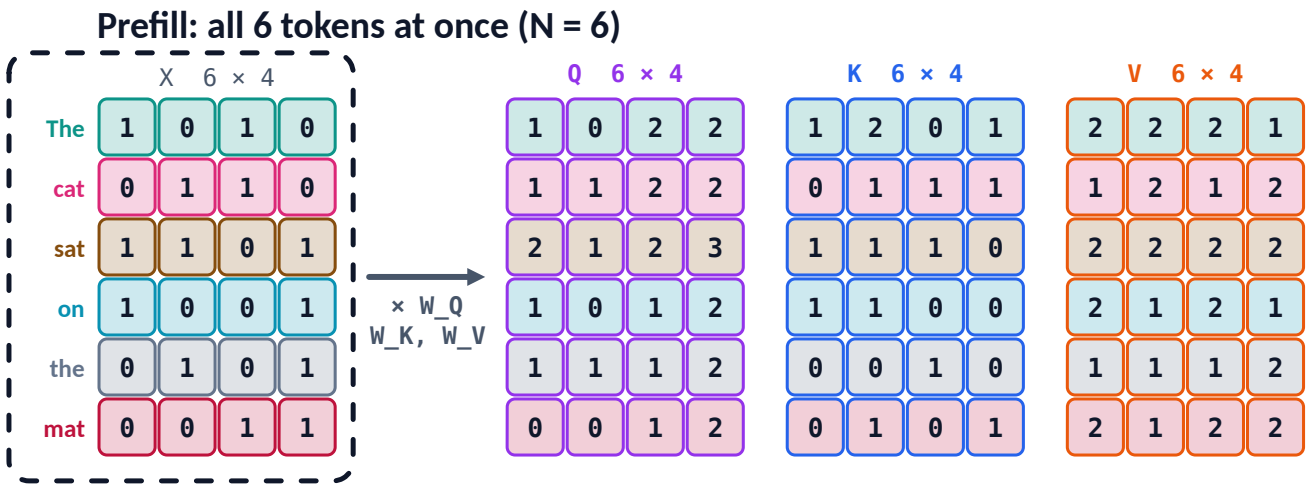
Prefill pushes N tokens through at once; decode pushes one.

Prefill: all 6 tokens at once (N = 6)



One forward pass

Prefill pushes N tokens through at once; decode pushes one.

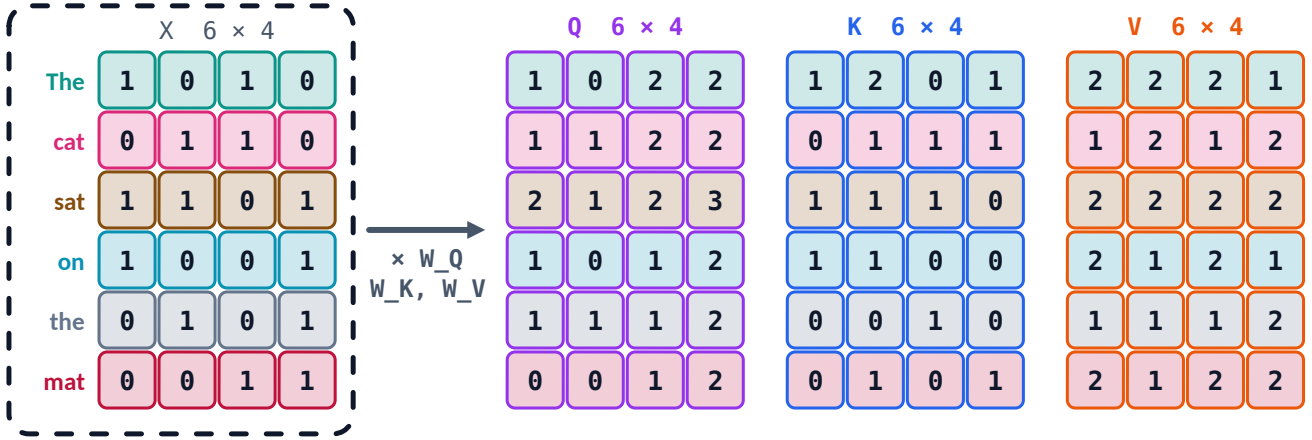


all 6 rows in one matmul, in parallel

One forward pass

Prefill pushes N tokens through at once; decode pushes one.

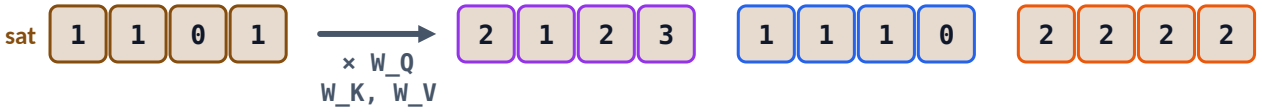
Prefill: all 6 tokens at once (N = 6)



all 6 rows in one matmul, in parallel

Decode: 1 token per step (N = 1)

(prompt: The cat)



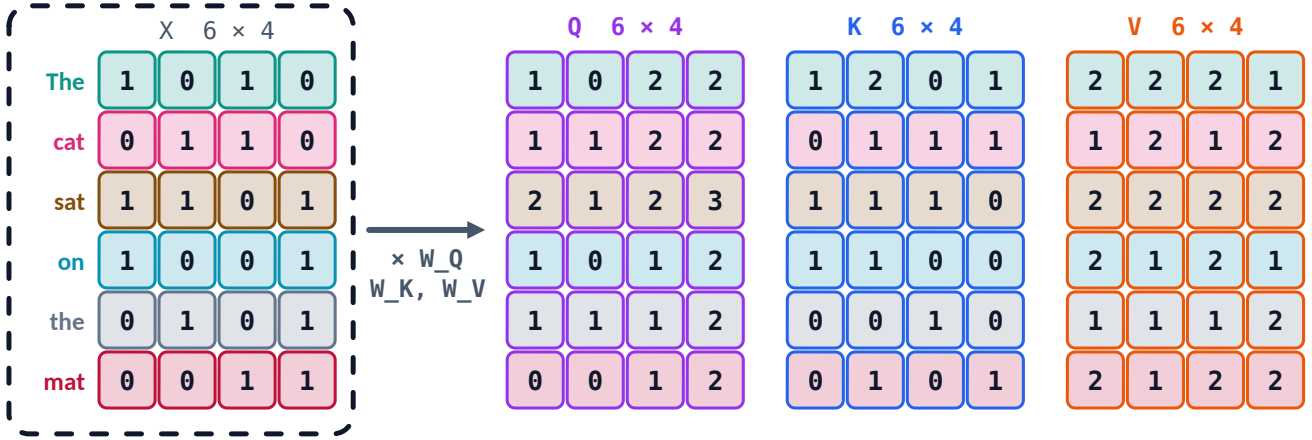
input = the token just sampled (sat)

its pass → logits for the next word

One forward pass

Prefill pushes N tokens through at once; decode pushes one.

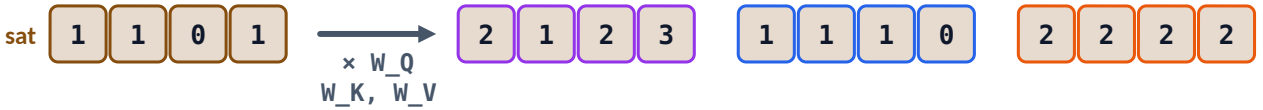
Prefill: all 6 tokens at once (N = 6)



all 6 rows in one matmul, in parallel

Decode: 1 token per step (N = 1)

(prompt: The cat)



input = the token just sampled (sat)

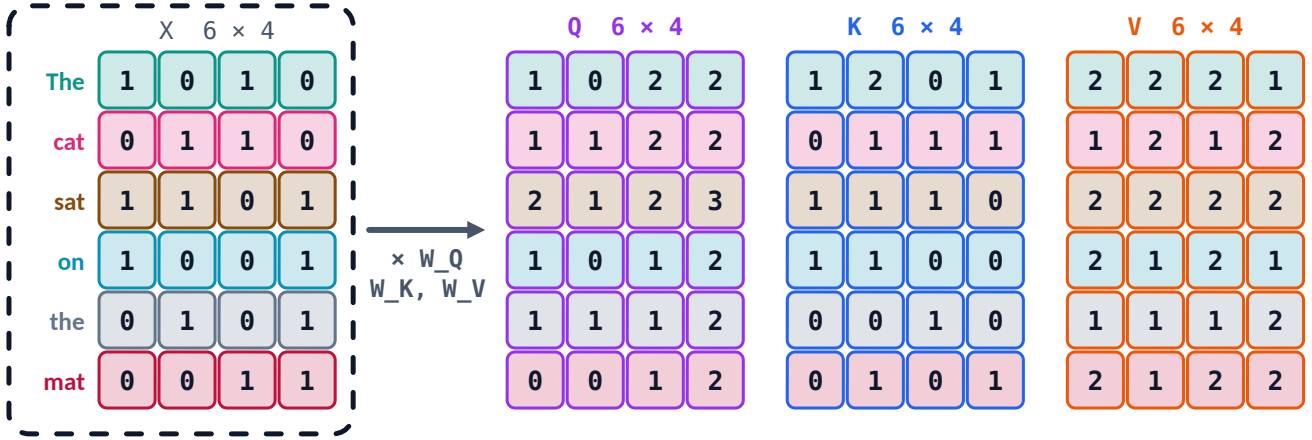
its pass → logits for the next word



One forward pass

Prefill pushes N tokens through at once; decode pushes one.

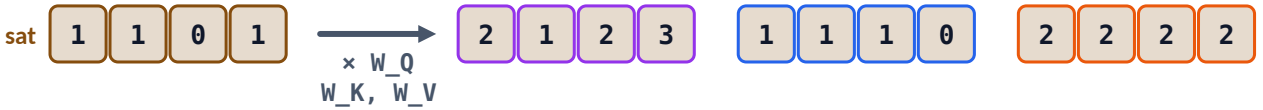
Prefill: all 6 tokens at once (N = 6)



all 6 rows in one matmul, in parallel

Decode: 1 token per step (N = 1)

(prompt: The cat)



input = the token just sampled (sat)

its pass → logits for the next word

real model: 64 layers = 16 attention + 48 other (linear attention)

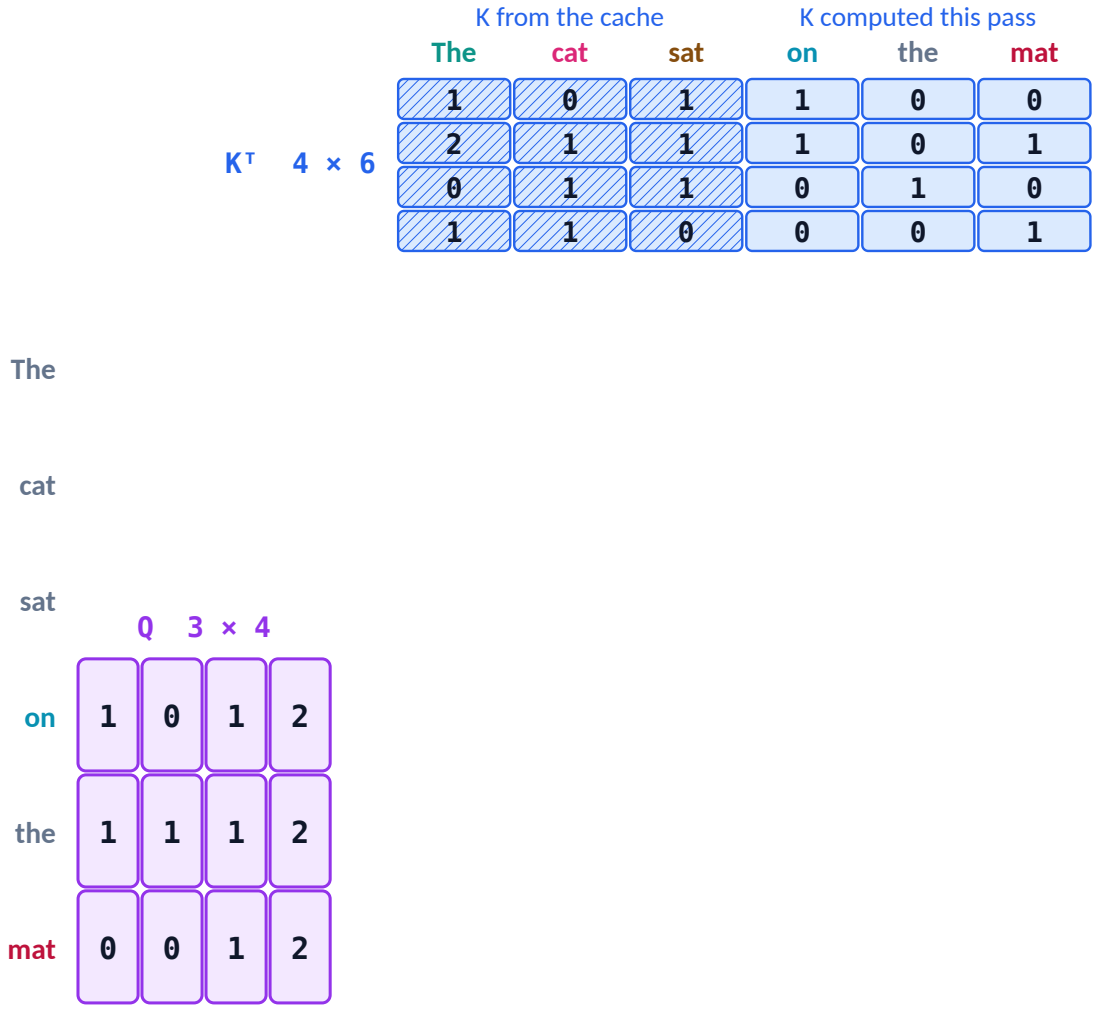


every layer: own weights

every attention layer: own k, v

Attention scores: Q · K^T

The sentence arrives in two chunks. This pass handles “on the mat”.



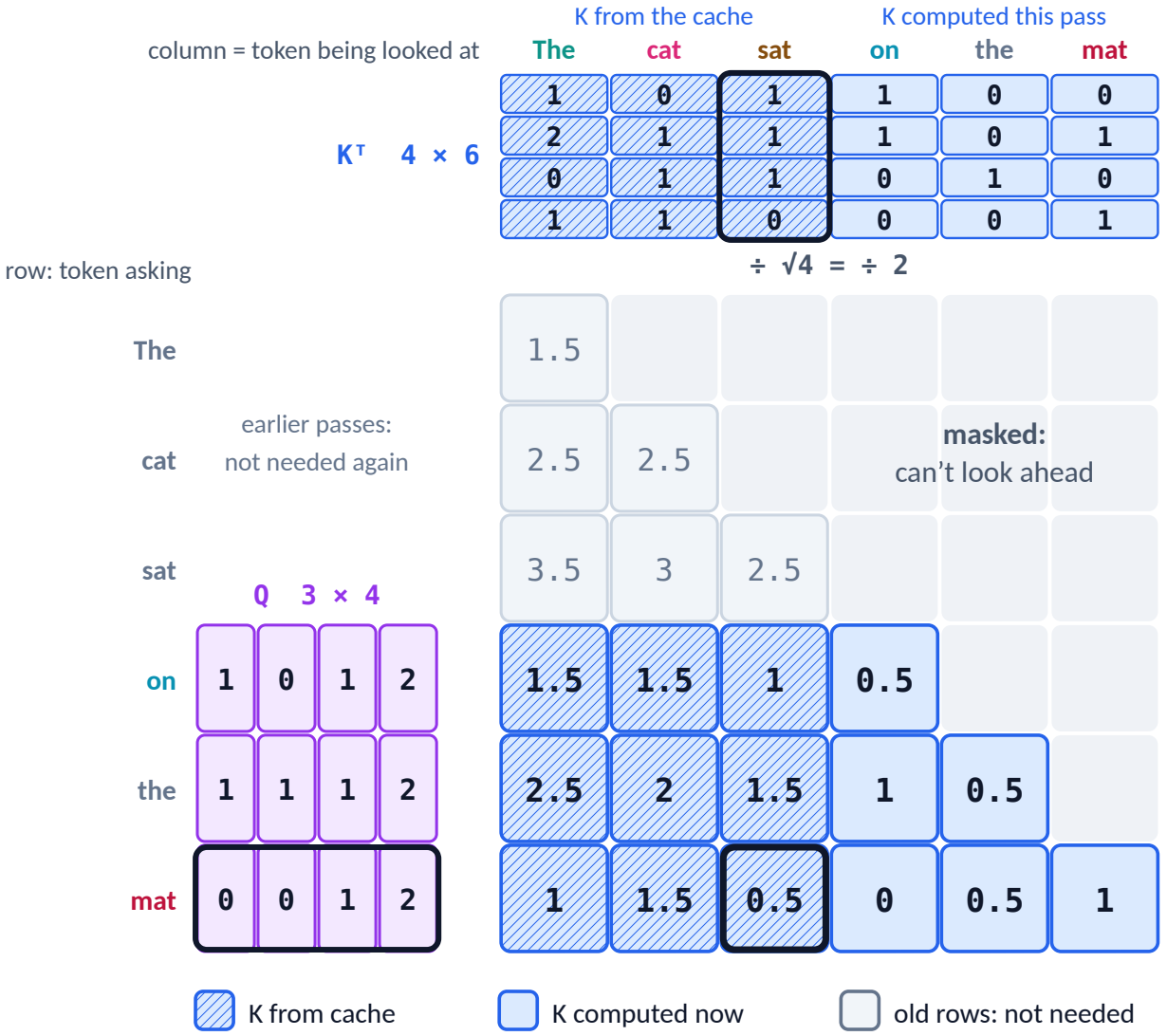
Attention scores: Q · K^T

The sentence arrives in two chunks. This pass handles “on the mat”.



Attention scores: Q · K^T

The sentence arrives in two chunks. This pass handles “on the mat”.



q_mat · k_sat

= 0·1 + 0·1 + 1·1 + 2·0 = 1

÷ 2 → 0.5

Attention scores: Q · Kᵀ

The sentence arrives in two chunks. This pass handles “on the mat”.

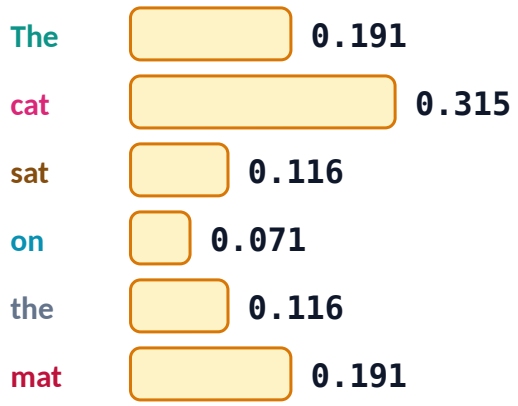


q_mat · k_sat

= 0·1 + 0·1 + 1·1 + 2·0 = 1

÷ 2 → 0.5

softmax of row “mat”

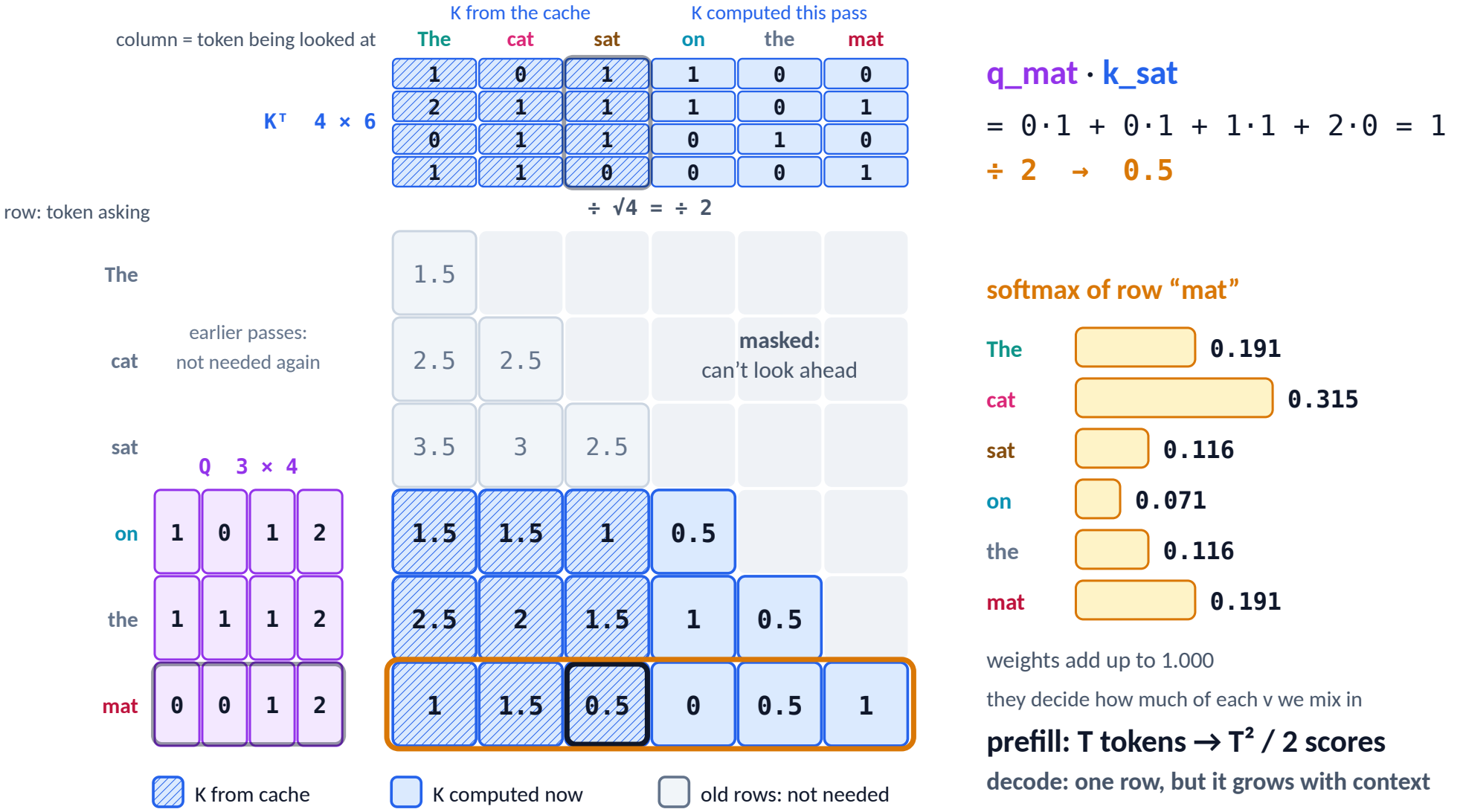


weights add up to 1.000

they decide how much of each v we mix in

Attention scores: Q · K^T

The sentence arrives in two chunks. This pass handles “on the mat”.



$q_{mat} \cdot k_{sat}$

$= 0 \cdot 1 + 0 \cdot 1 + 1 \cdot 1 + 2 \cdot 0 = 1$

$\div 2 \rightarrow 0.5$

softmax of row “mat”

The	0.191
cat	0.315
sat	0.116
on	0.071
the	0.116
mat	0.191

weights add up to 1.000

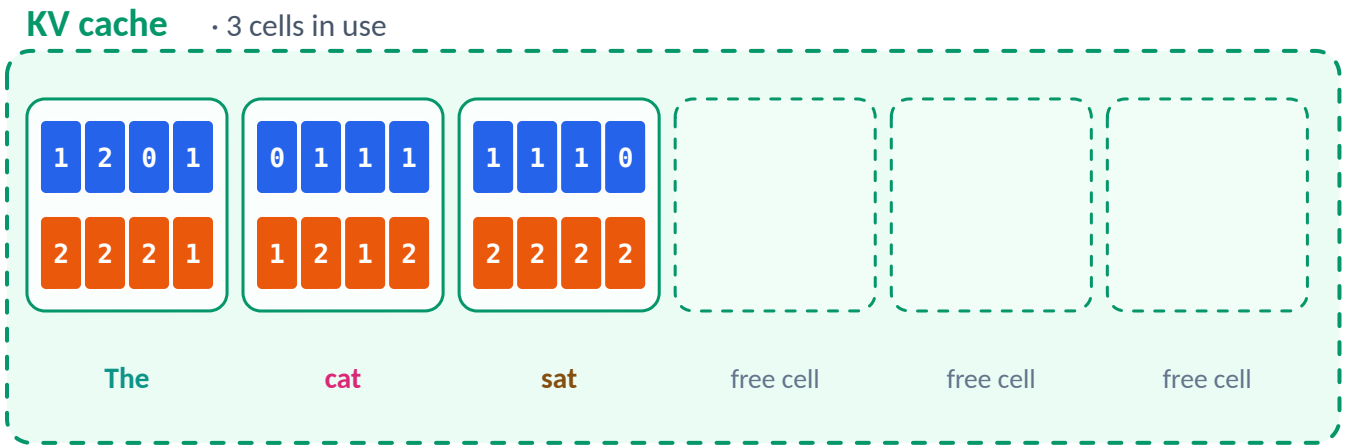
they decide how much of each v we mix in

prefill: T tokens → T^2 / 2 scores

decode: one row, but it grows with context

What must be remembered

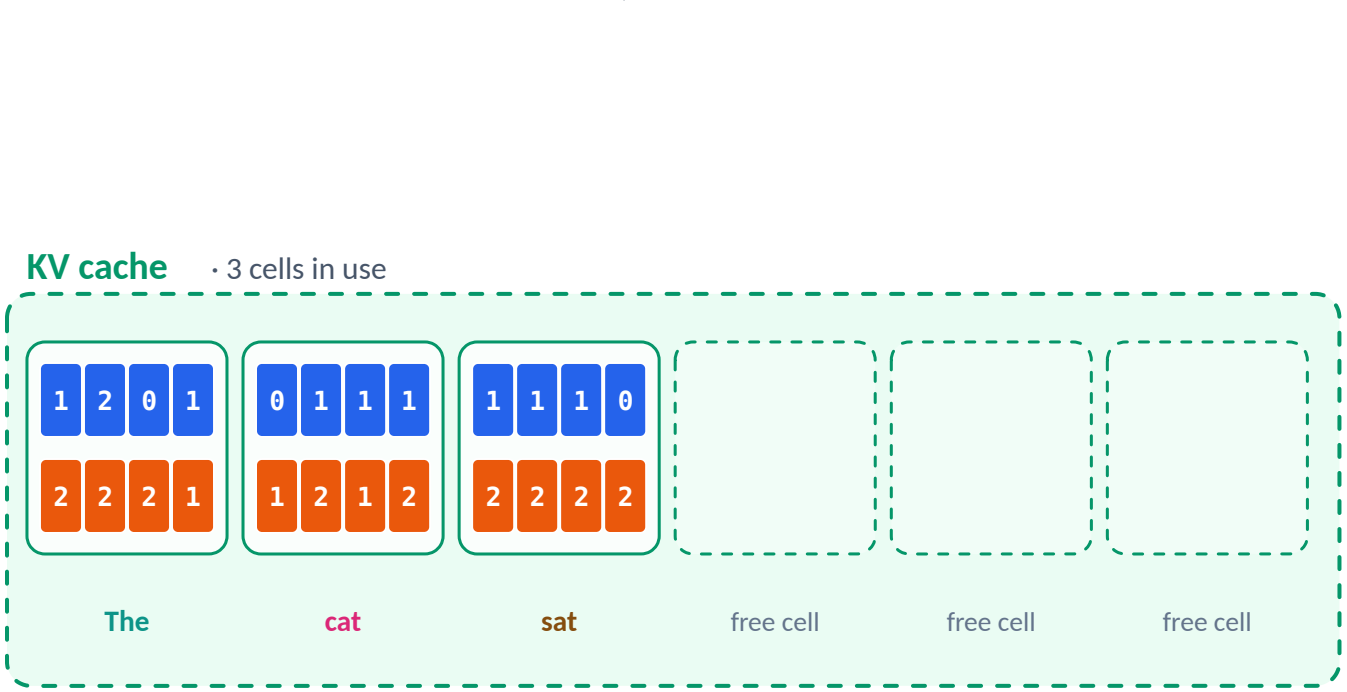
A different moment: three tokens cached, the next chunk arrives.

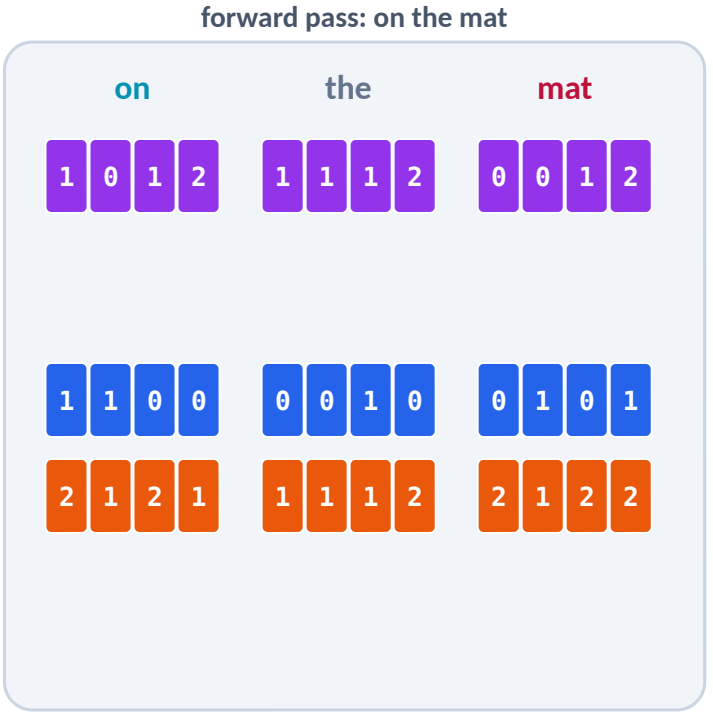


1 cell = the k and v of one token

What must be remembered

A different moment: three tokens cached, the next chunk arrives.

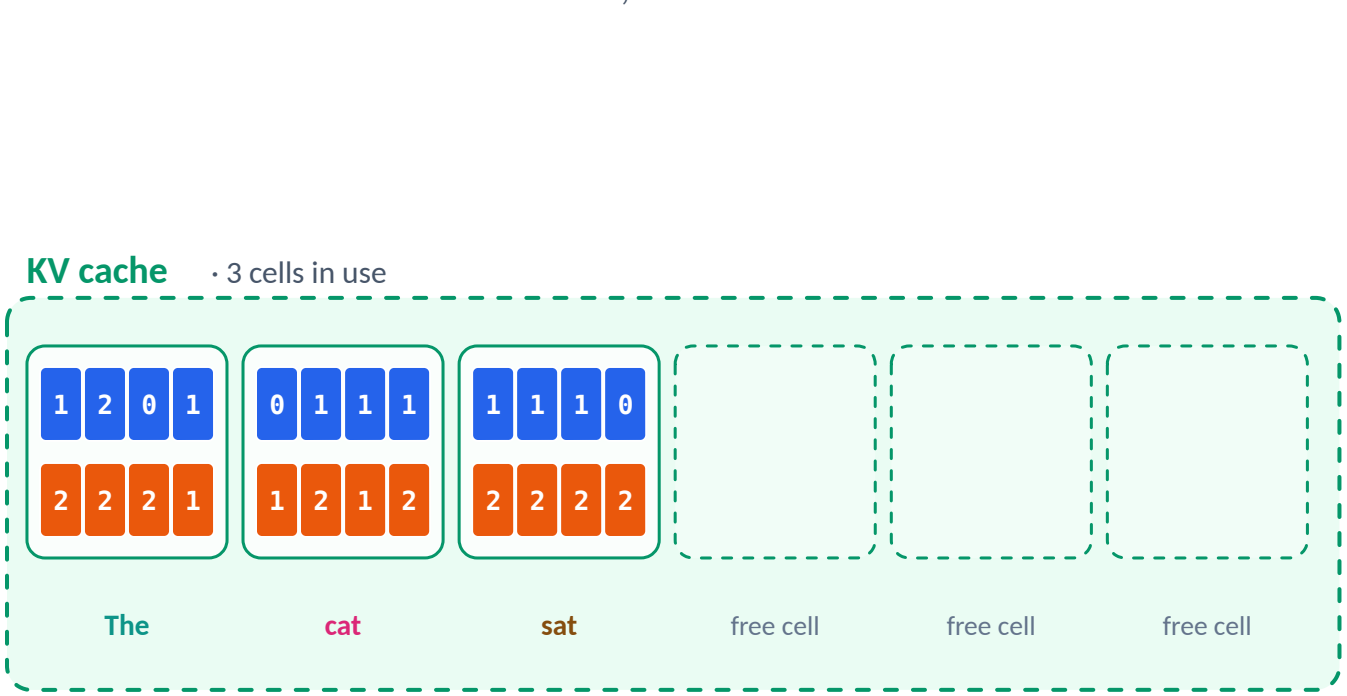


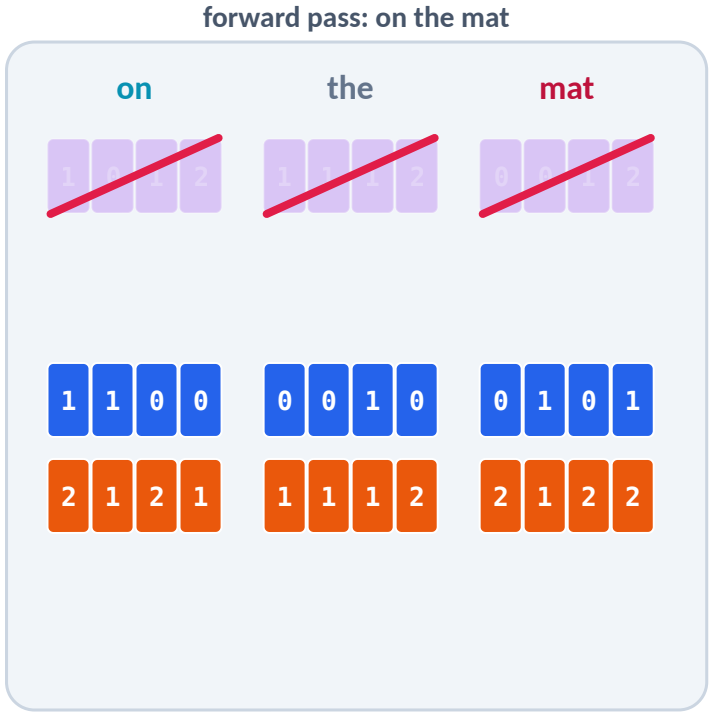


new tokens: q, k, v computed

What must be remembered

A different moment: three tokens cached, the next chunk arrives.

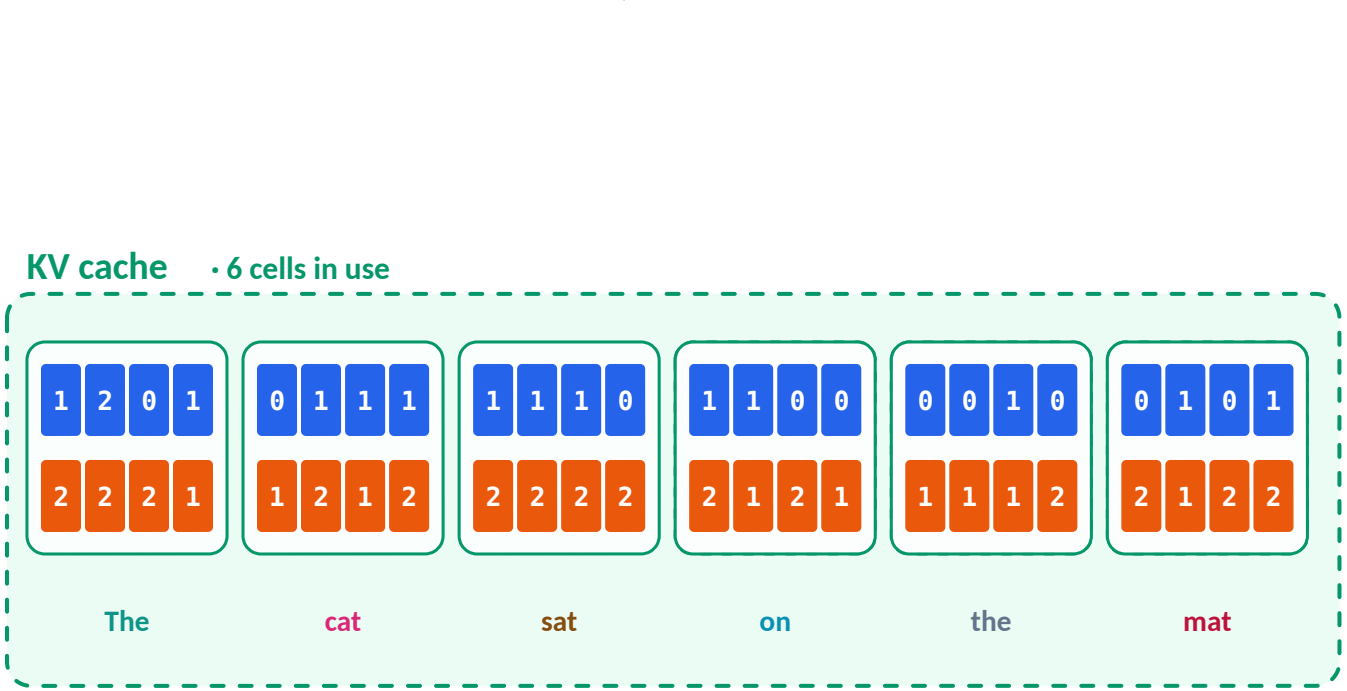


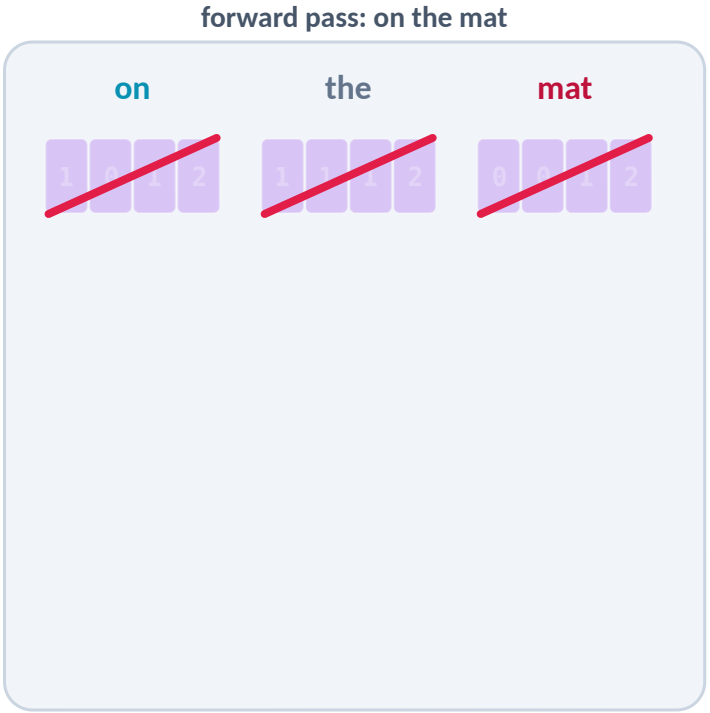


q: used in this pass only → dropped

What must be remembered

A different moment: three tokens cached, the next chunk arrives.

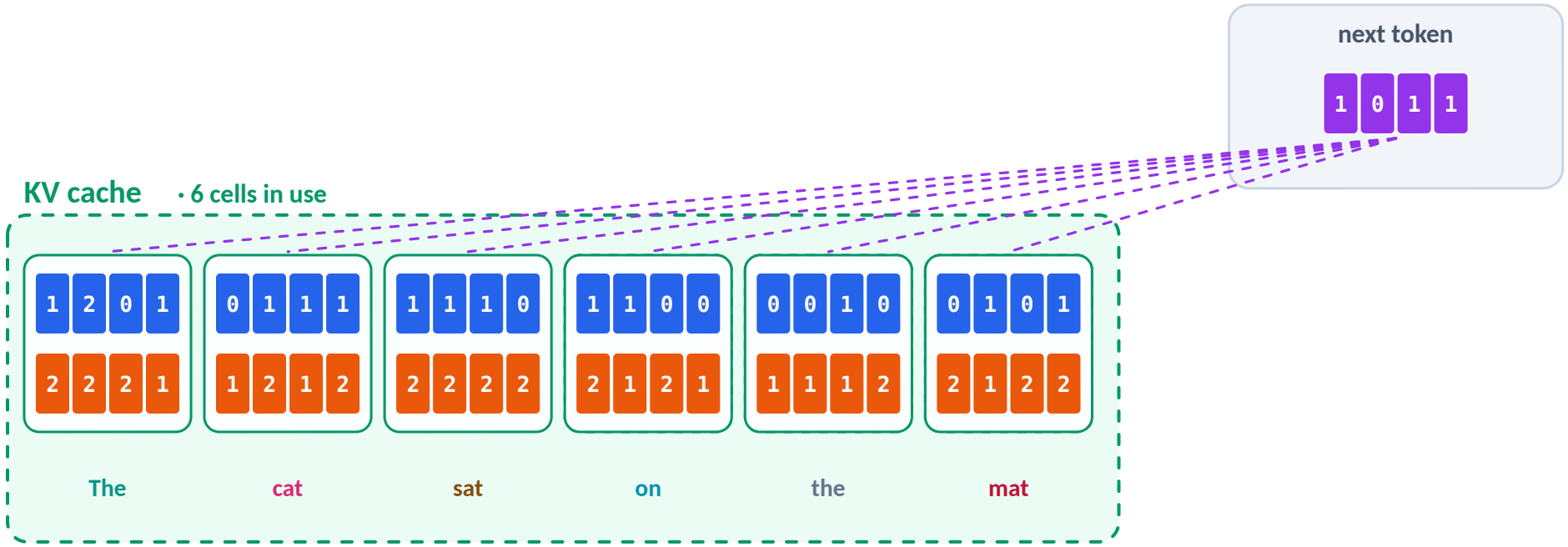




k, v → free cells: cache grows 3 → 6

What must be remembered

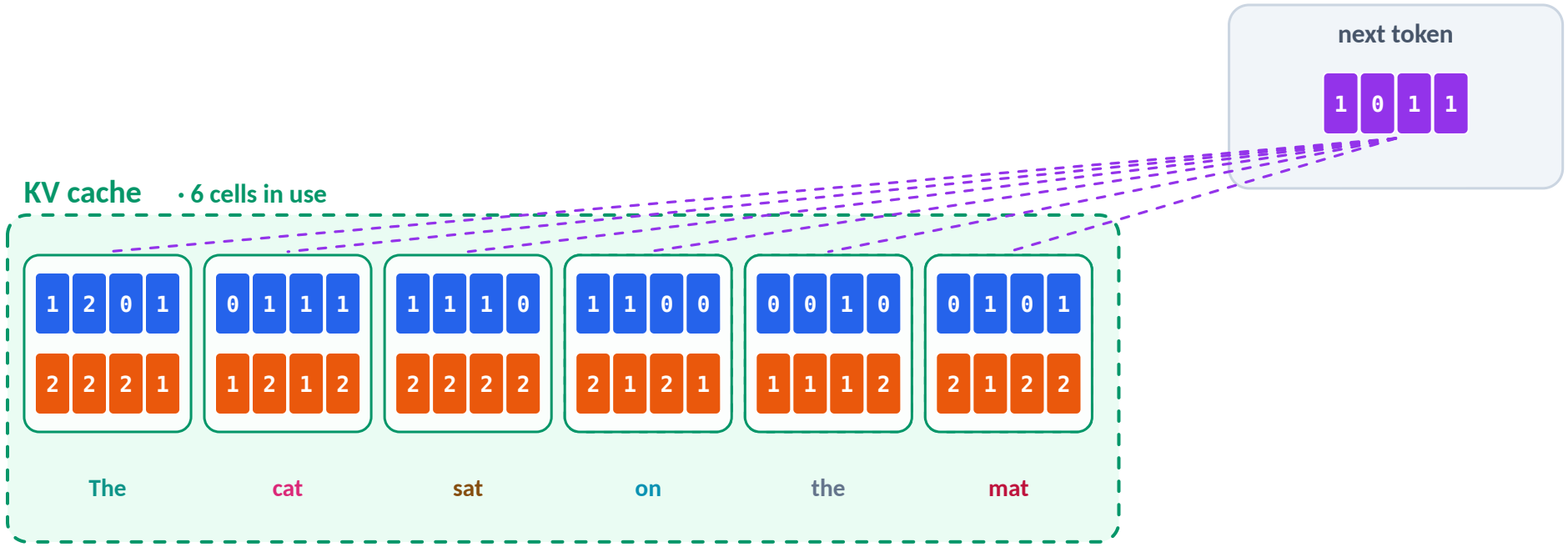
A different moment: three tokens cached, the next chunk arrives.



next token: all 6 cells + itself

What must be remembered

A different moment: three tokens cached, the next chunk arrives.

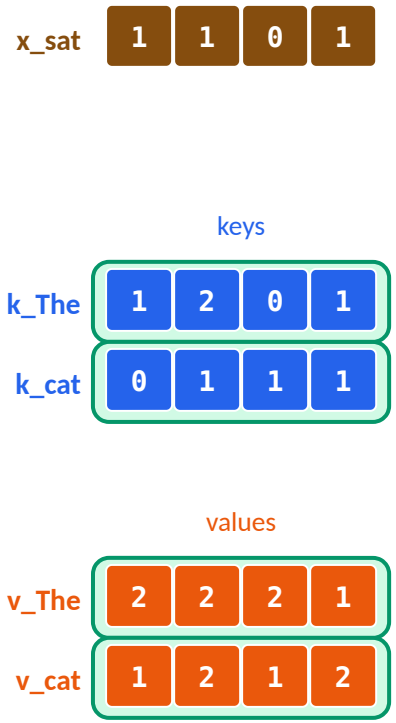


we trade memory for compute

the price is memory: that is the KV cache

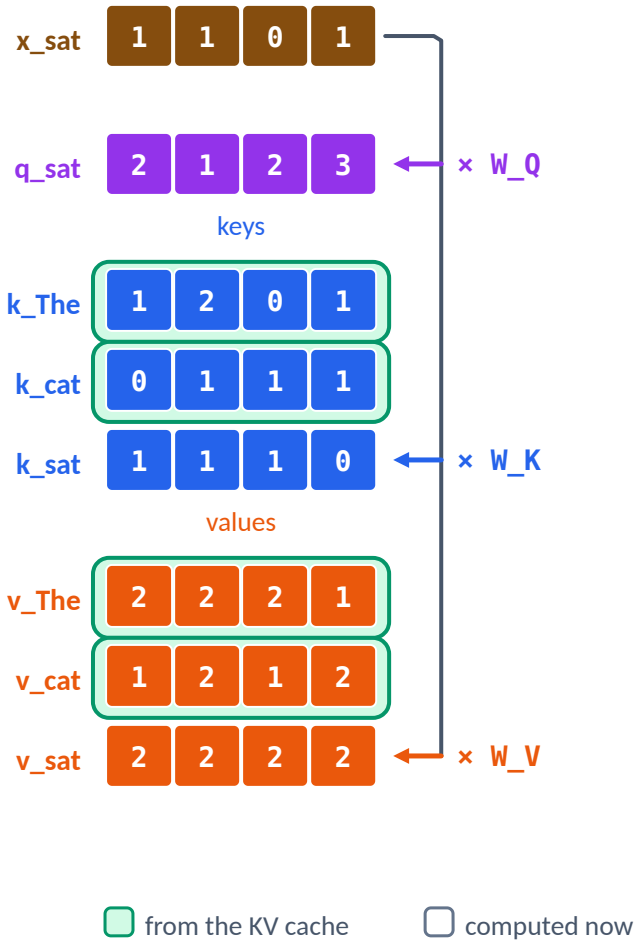
One token’s attention, start to finish

Rewind to when “sat” was processed: its q meets the cached keys.



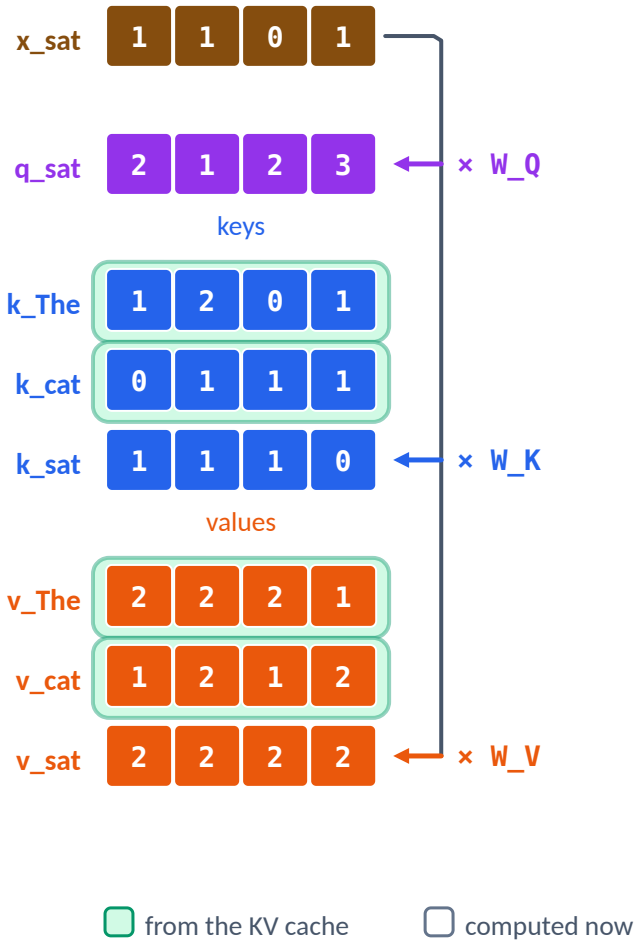
One token’s attention, start to finish

Rewind to when “sat” was processed: its q meets the cached keys.



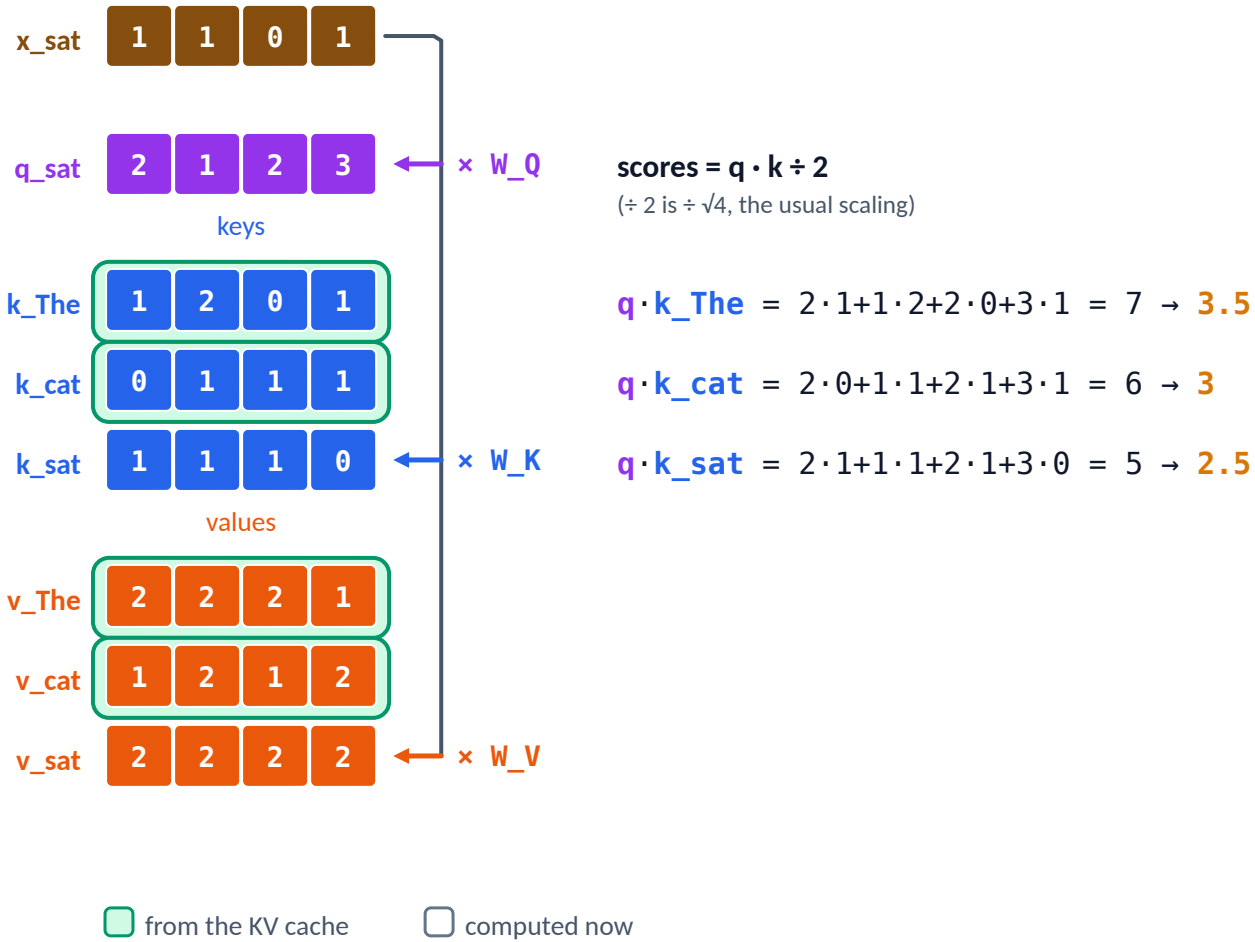
One token’s attention, start to finish

Rewind to when “sat” was processed: its q meets the cached keys.



One token’s attention, start to finish

Rewind to when “sat” was processed: its q meets the cached keys.



scores = q · k ÷ 2

(÷ 2 is ÷ √4, the usual scaling)

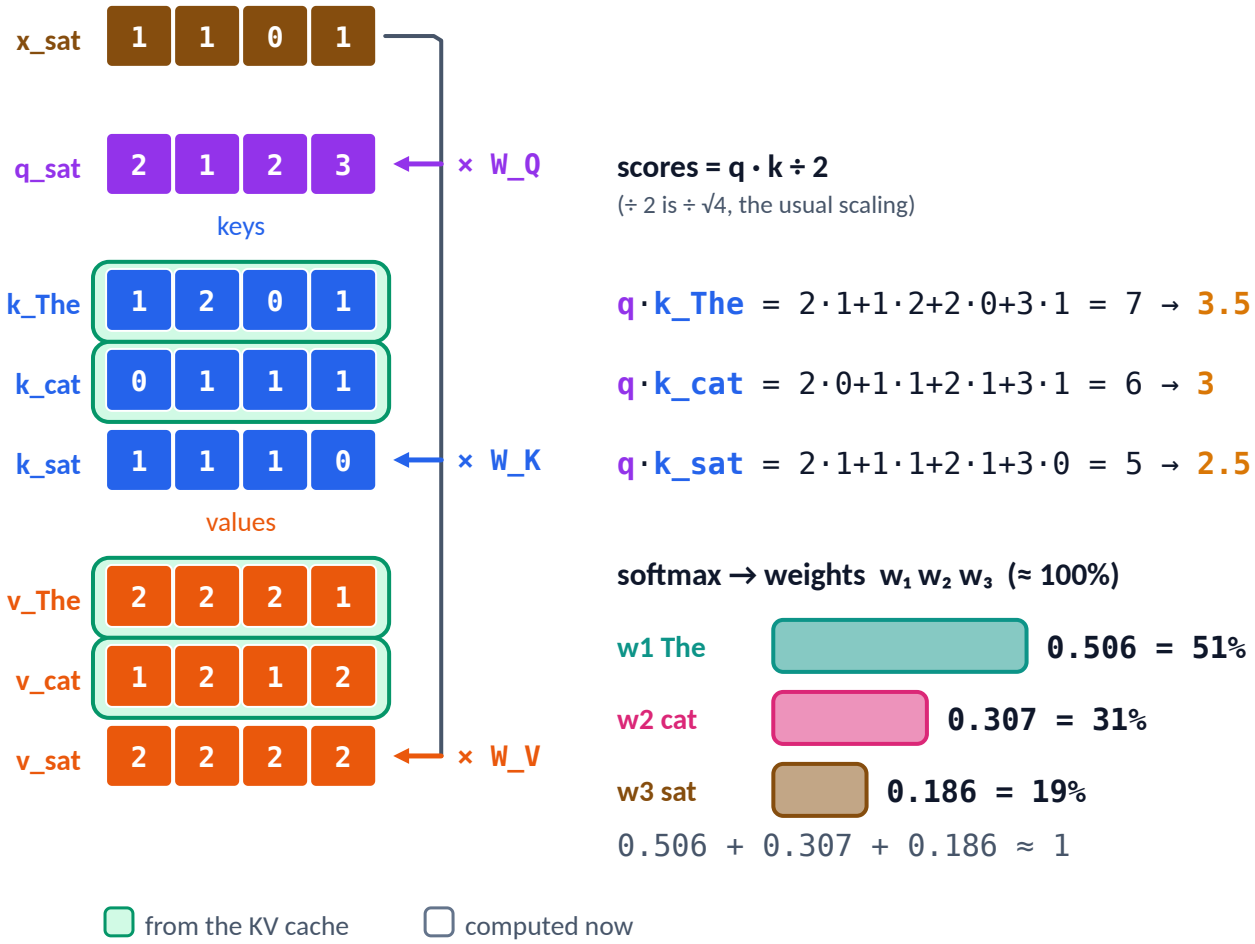
`q·k_The = 2·1+1·2+2·0+3·1 = 7 → 3.5`

`q·k_cat = 2·0+1·1+2·1+3·1 = 6 → 3`

`q·k_sat = 2·1+1·1+2·1+3·0 = 5 → 2.5`

One token’s attention, start to finish

Rewind to when “sat” was processed: its q meets the cached keys.



scores = q · k ÷ 2

(÷ 2 is ÷ √4, the usual scaling)

`q·k_The = 2·1+1·2+2·0+3·1 = 7 → 3.5`

`q·k_cat = 2·0+1·1+2·1+3·1 = 6 → 3`

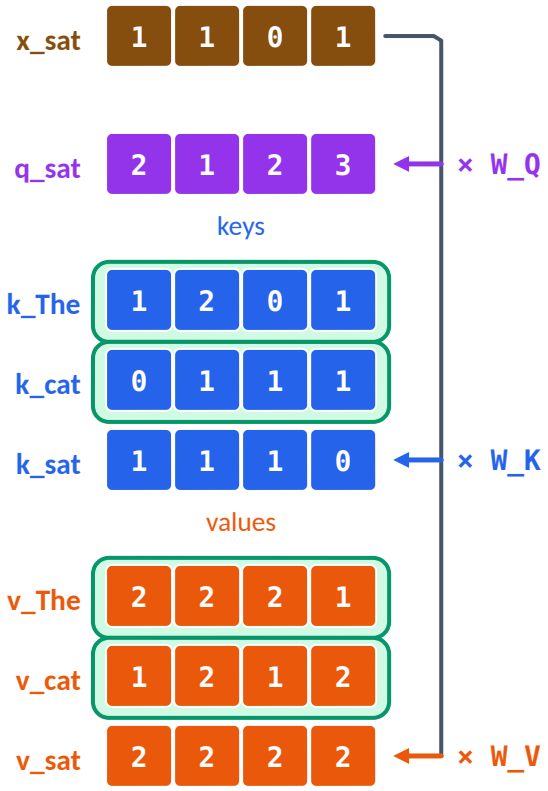
`q·k_sat = 2·1+1·1+2·1+3·0 = 5 → 2.5`

softmax → weights $w_1\ w_2\ w_3$ (≈ 100%)

`0.506 + 0.307 + 0.186 ≈ 1`

One token's attention, start to finish

Rewind to when "sat" was processed: its q meets the cached keys.



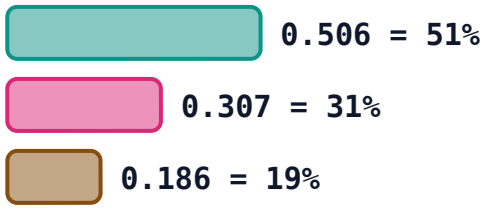
scores = q · k ÷ 2
(÷ 2 is ÷ √4, the usual scaling)

`q·k_The = 2·1+1·2+2·0+3·1 = 7 → 3.5`

`q·k_cat = 2·0+1·1+2·1+3·1 = 6 → 3`

`q·k_sat = 2·1+1·1+2·1+3·0 = 5 → 2.5`

softmax → weights $w_1\ w_2\ w_3$ (≈ 100%)

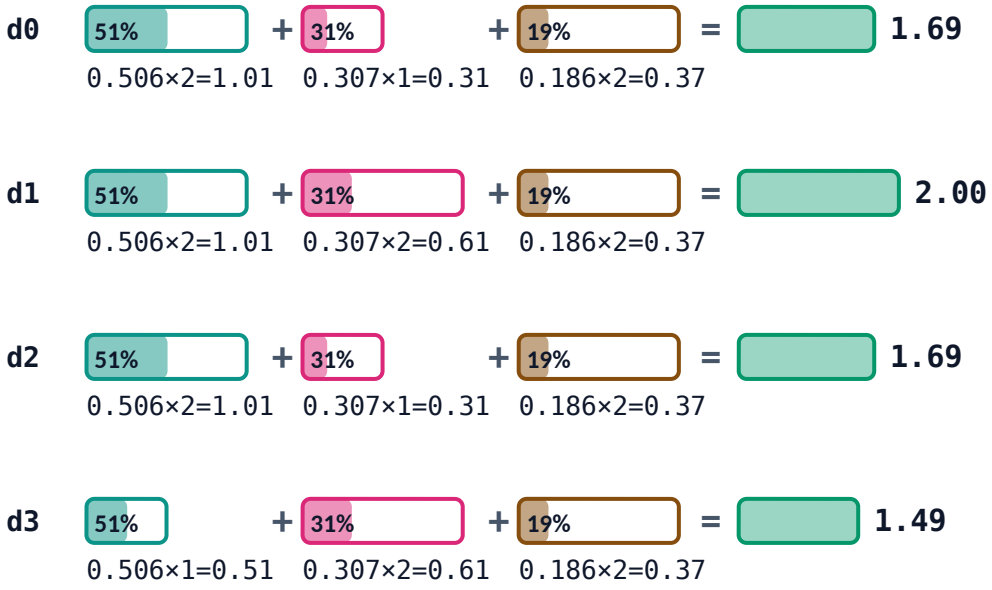


`0.506 + 0.307 + 0.186 ≈ 1`

from the KV cache computed now

out[d] = w_1·v_The[d] + w_2·v_cat[d] + w_3·v_sat[d]

same percents in every dimension; only the values being taken change

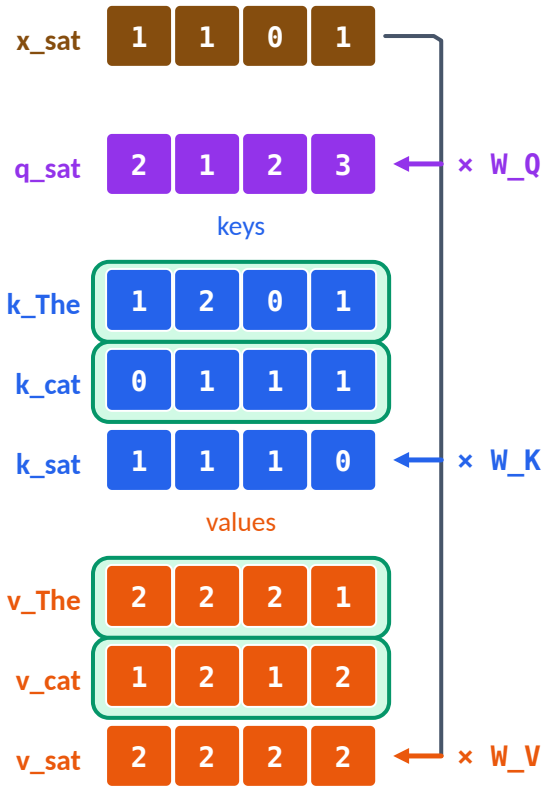


outline = the token's value · filled part = what we take of it

The cat sat

One token's attention, start to finish

Rewind to when "sat" was processed: its q meets the cached keys.

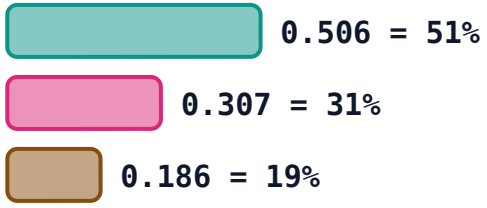


scores = q · k ÷ 2

(÷ 2 is ÷ √4, the usual scaling)

```
q·k_The = 2·1+1·2+2·0+3·1 = 7 → 3.5
q·k_cat = 2·0+1·1+2·1+3·1 = 6 → 3
q·k_sat = 2·1+1·1+2·1+3·0 = 5 → 2.5
```

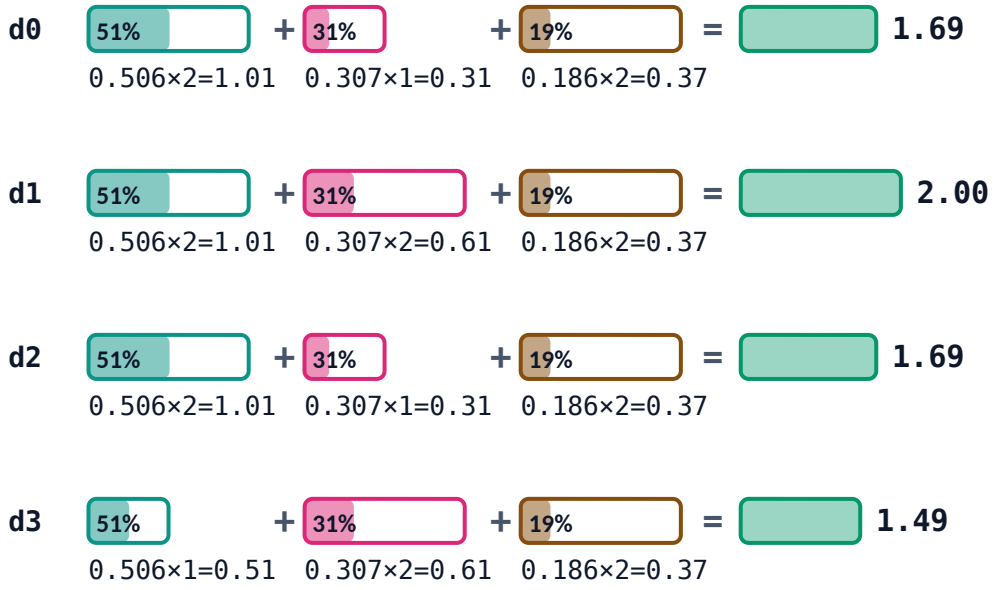
softmax → weights $w_1\ w_2\ w_3$ (≈ 100%)



`0.506 + 0.307 + 0.186 ≈ 1`

out[d] = w_1·v_The[d] + w_2·v_cat[d] + w_3·v_sat[d]

same percents in every dimension; only the values being taken change



outline = the token's value · filled part = what we take of it

The cat sat



contextualized "sat"

→ goes on to the next layer

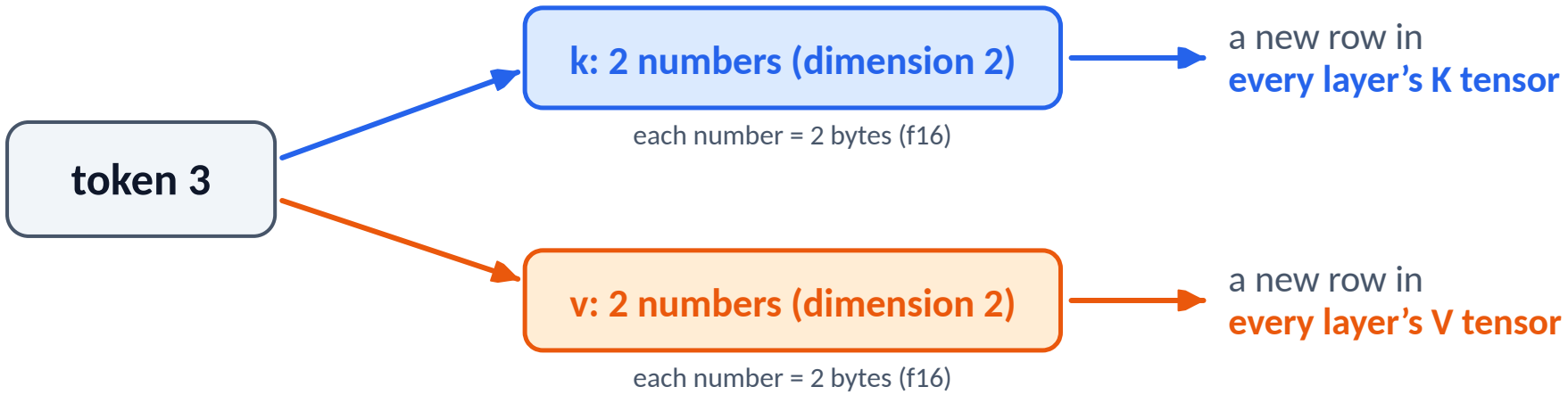
The KV cache is just tensors in GPU memory

Toy: each token's k and v have 2 numbers (dimension 2), f16 = 2 bytes per number, 6 cells, 2 layers

each tensor: 6 cells × 2 numbers × 2 B = 24 B

GPU buffer · toy base address 0x1000 · address = base + byte offset

K0	V0	K1	V1
6 vectors × 2 numbers × 2 B = 24 B	6 vectors × 2 numbers × 2 B = 24 B	6 vectors × 2 numbers × 2 B = 24 B	6 vectors × 2 numbers × 2 B = 24 B



Each layer keeps one K tensor and one V tensor: one row per token

The KV cache is just tensors in GPU memory

Toy: each token's k and v have 2 numbers (dimension 2), f16 = 2 bytes per number, 6 cells, 2 layers

each tensor: 6 cells × 2 numbers × 2 B = 24 B

GPU buffer · toy base address 0x1000 · address = base + byte offset



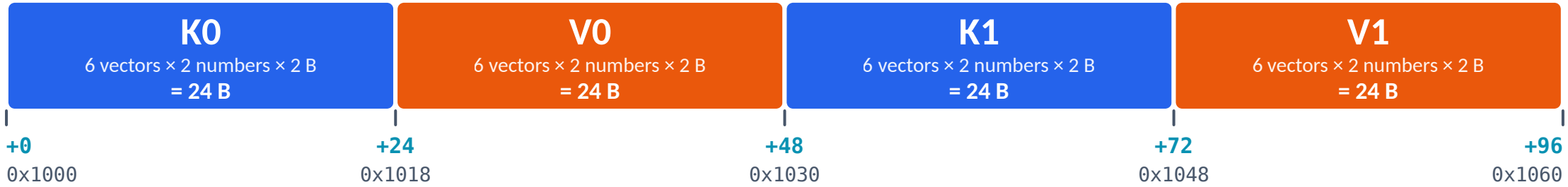
`real model, 4,096 cells: 8 MiB per tensor · row = 2,048 B · 64 KiB per cell`

The KV cache is just tensors in GPU memory

Toy: each token's k and v have 2 numbers (dimension 2), f16 = 2 bytes per number, 6 cells, 2 layers

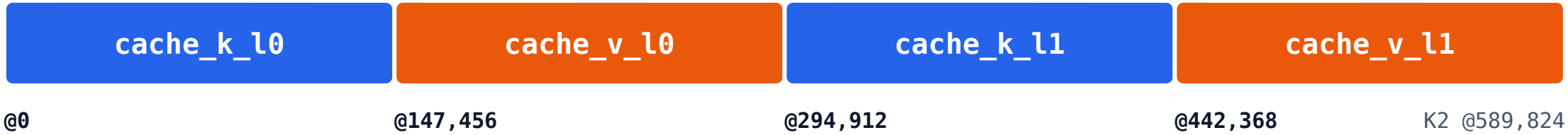
each tensor: 6 cells × 2 numbers × 2 B = 24 B

GPU buffer · toy base address 0x1000 · address = base + byte offset



✓ **Checked on a real run: K0 V0 K1 V1, back to back — K0 and K1 are not neighbours**

the real run: stories15M, CPU backend, tensor offsets in bytes



each tensor is 147,456 B, so no padding: sizes are multiples of the 32 B CPU alignment.
CUDA aligns to 128 B (read from the source, not run): the same layout, as sizes here are multiples of it too.

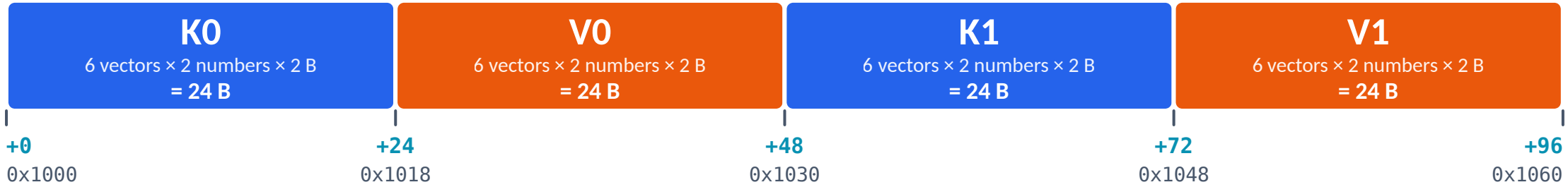
`real model, 4,096 cells: 8 MiB per tensor · row = 2,048 B · 64 KiB per cell`

The KV cache is just tensors in GPU memory

Toy: each token's k and v have 2 numbers (dimension 2), f16 = 2 bytes per number, 6 cells, 2 layers

each tensor: 6 cells × 2 numbers × 2 B = 24 B

GPU buffer · toy base address 0x1000 · address = base + byte offset



	number 0 (2 B)	number 1 (2 B)	row address
cell 0	+0	+2	0x1000
cell 1	+4	+6	0x1004
cell 2	+8	+10	0x1008
cell 3	+12	+14	0x100C
cell 4	+16	+18	0x1010
cell 5	+20	+22	0x1014

1 row = 1 cell = 4 B

K0: 6 rows, one per cell

one row = one token's k vector

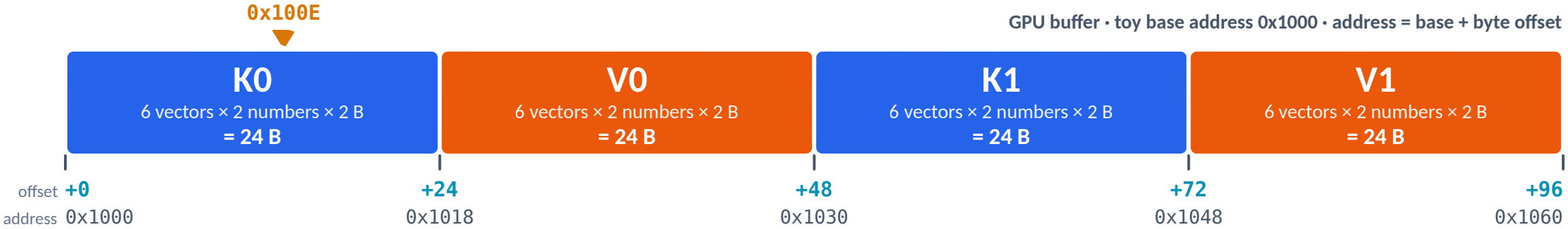
2 numbers (dimension 2) × 2 B = 4 B

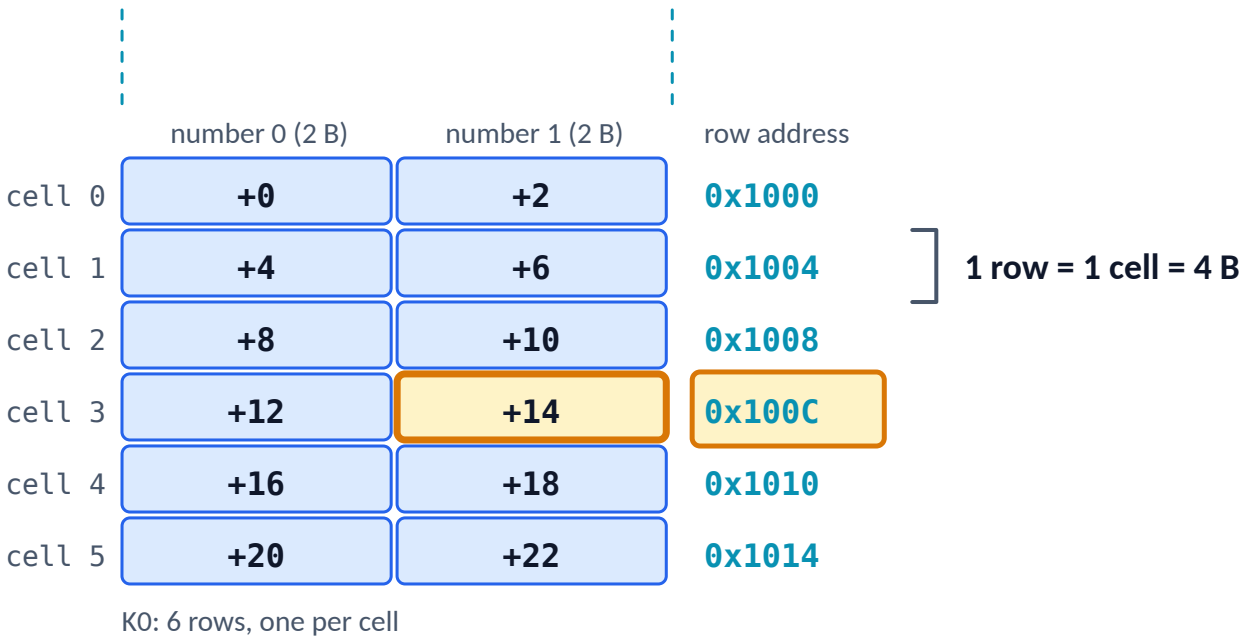
K0 = 6 rows × 4 B = 24 B

`real model, 4,096 cells: 8 MiB per tensor · row = 2,048 B · 64 KiB per cell`

The KV cache is just tensors in GPU memory

Toy: each token's k and v have 2 numbers (dimension 2), f16 = 2 bytes per number, 6 cells, 2 layers





cell 3, number 1

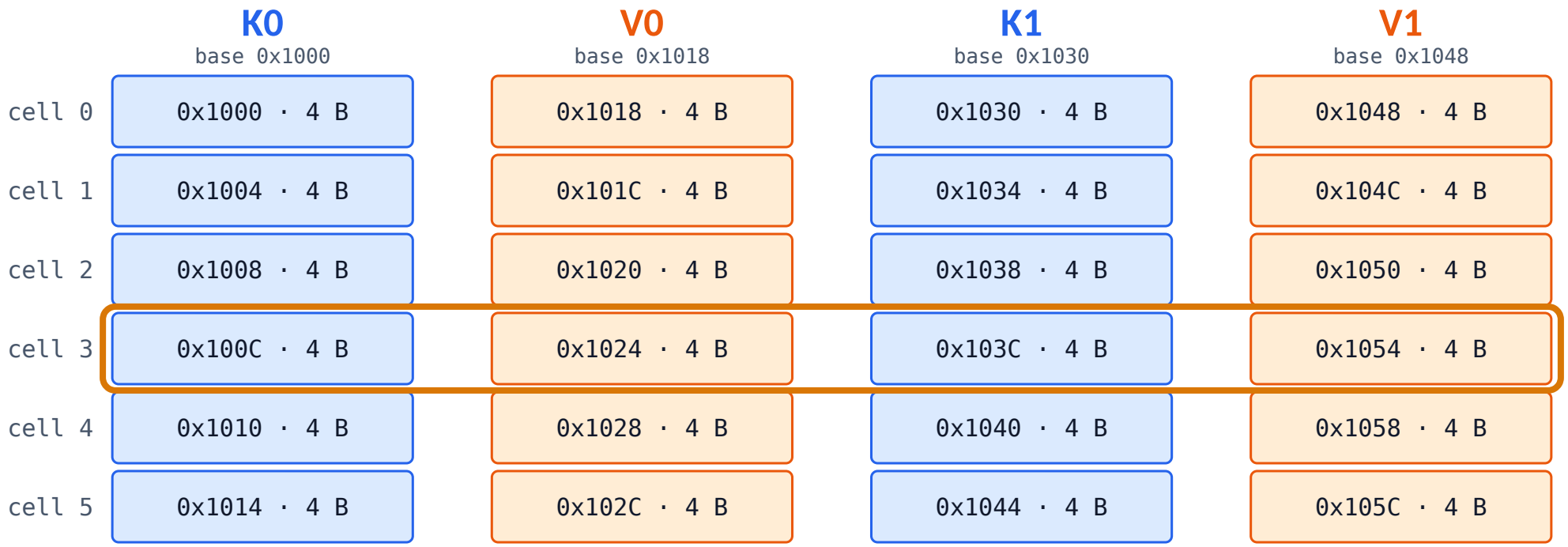
offset = 3×4 + 1×2 = 14

address = 0x1000 + 14 = 0x100E

real model, 4,096 cells: 8 MiB per tensor · row = 2,048 B · 64 KiB per cell

The KV cache is just tensors in GPU memory

Toy: each token's k and v have 2 numbers (dimension 2), f16 = 2 bytes per number, 6 cells, 2 layers



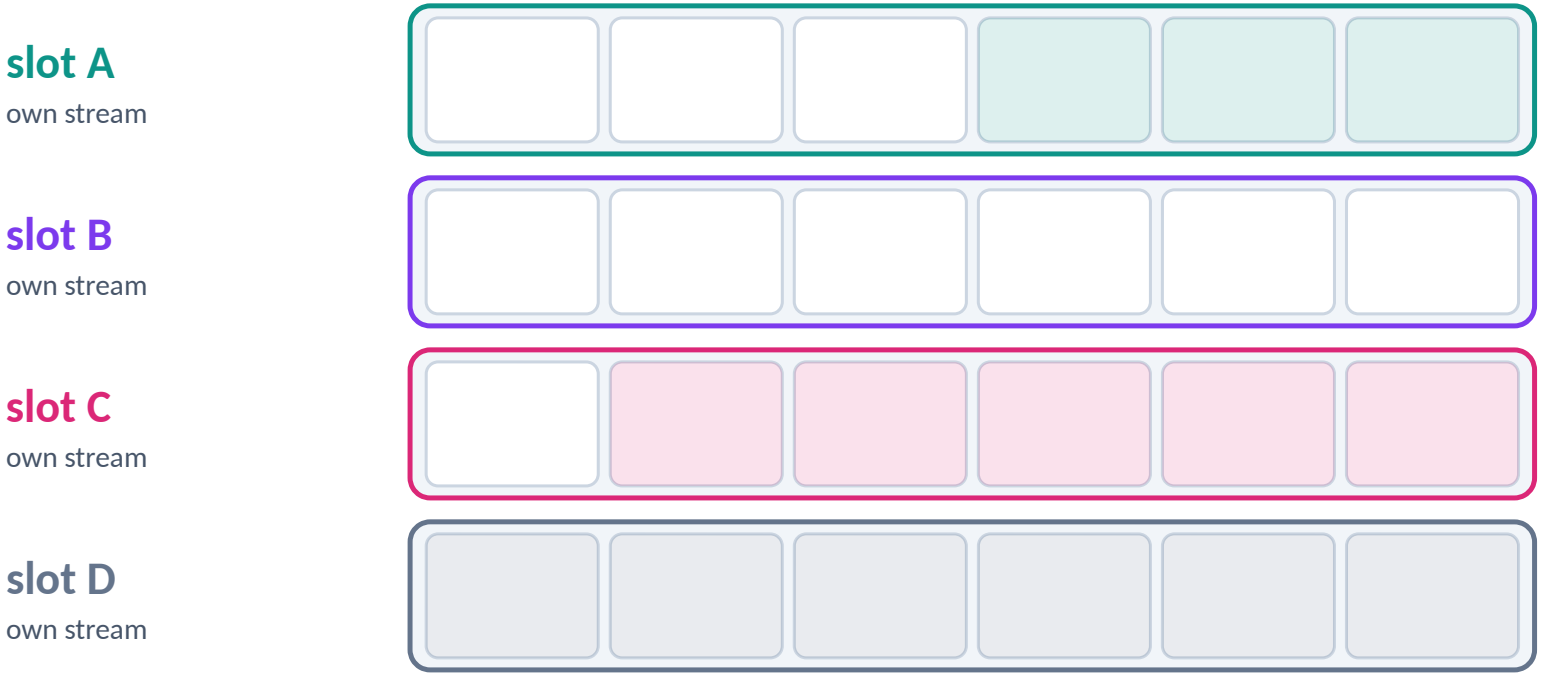
Cell 3 = one row in every K and V tensor, at the same offset (+12)

`0x100C · 0x1024 · 0x103C · 0x1054 (four places, a tensor apart: 24 B)`

`real model, 4,096 cells: 8 MiB per tensor · row = 2,048 B · 64 KiB per cell`

Split KV: one slab per slot

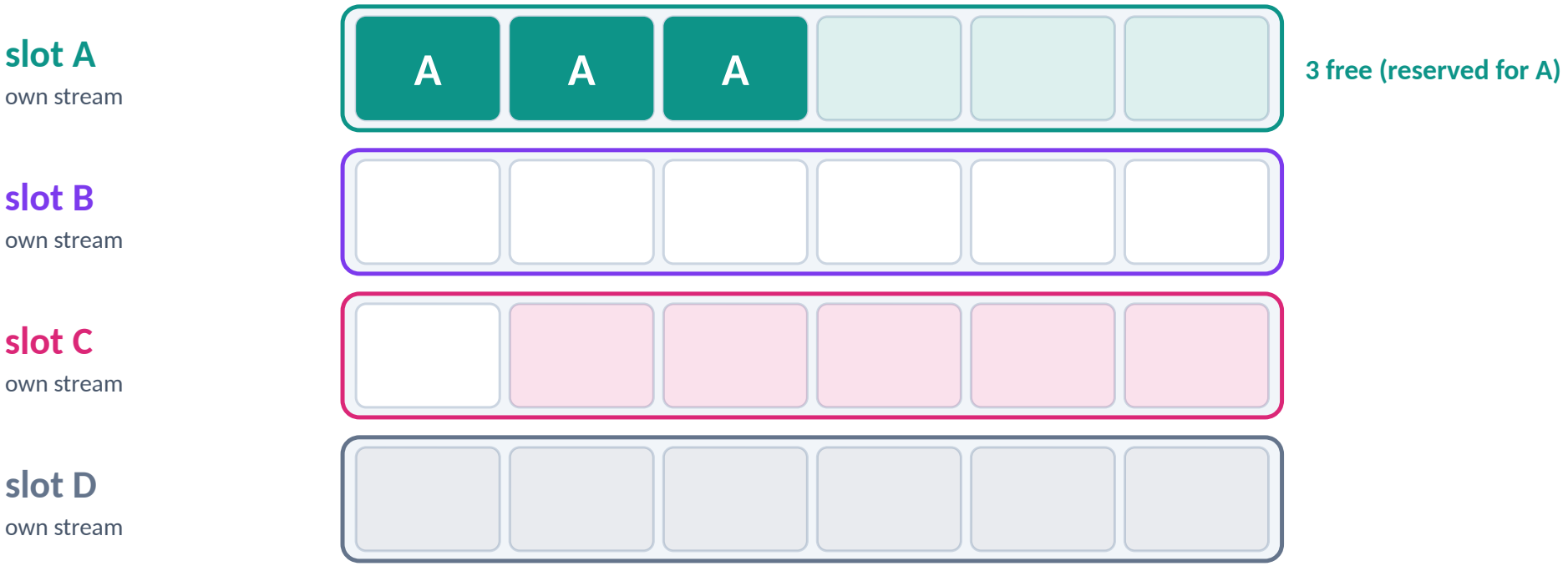
Non-unified: every slot owns a fixed slab of cells



Each slot owns a fixed slab

Split KV: one slab per slot

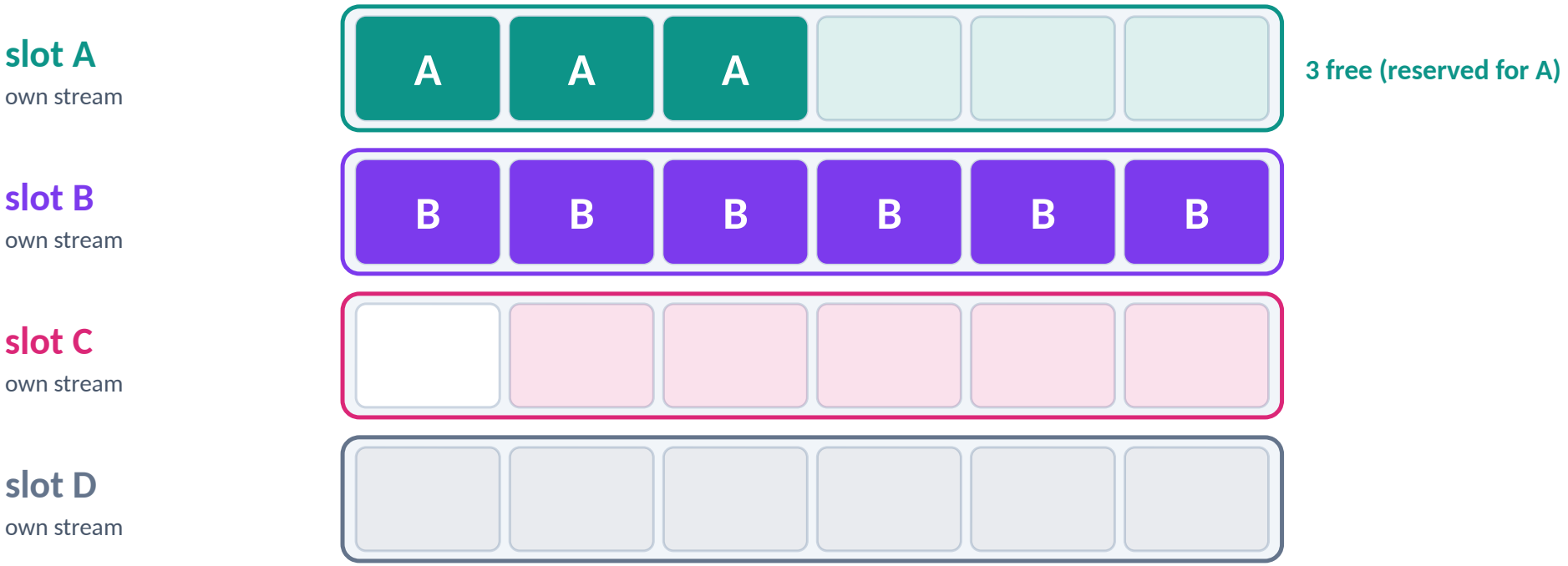
Non-unified: every slot owns a fixed slab of cells



Each slot owns a fixed slab

Split KV: one slab per slot

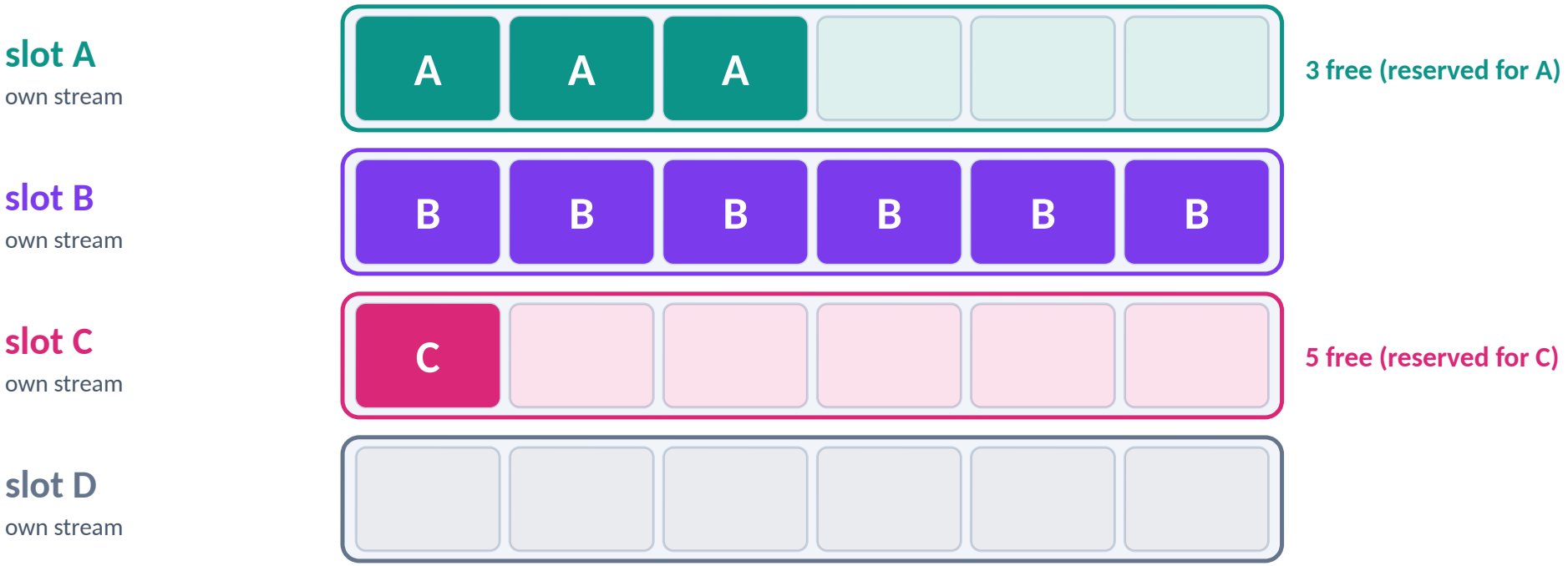
Non-unified: every slot owns a fixed slab of cells



Each slot owns a fixed slab

Split KV: one slab per slot

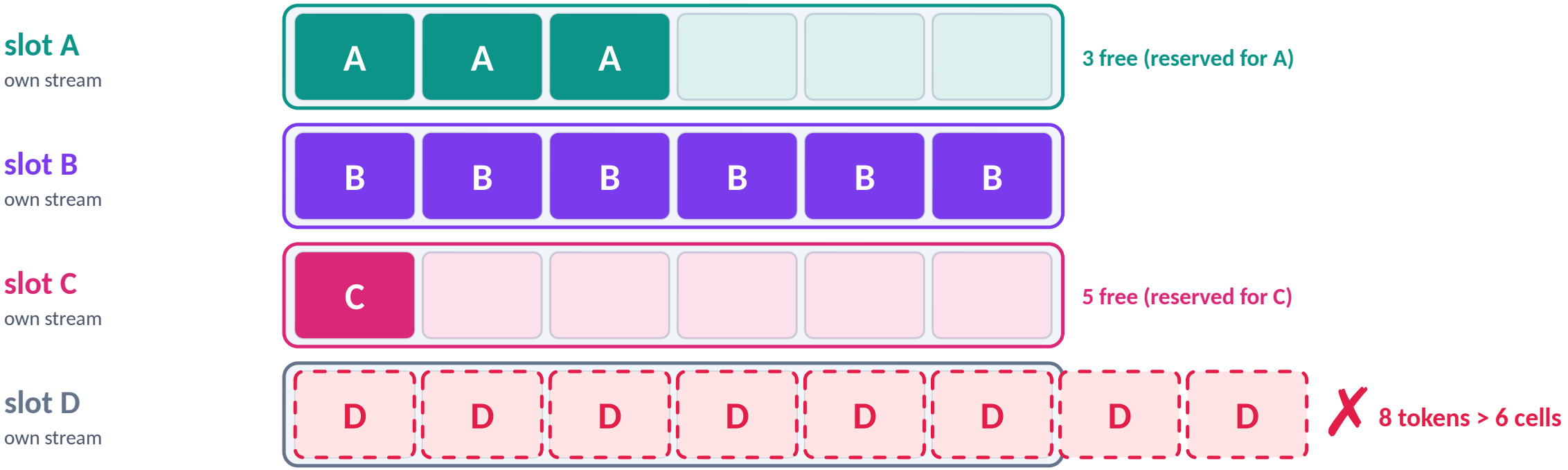
Non-unified: every slot owns a fixed slab of cells



Each slot owns a fixed slab

Split KV: one slab per slot

Non-unified: every slot owns a fixed slab of cells

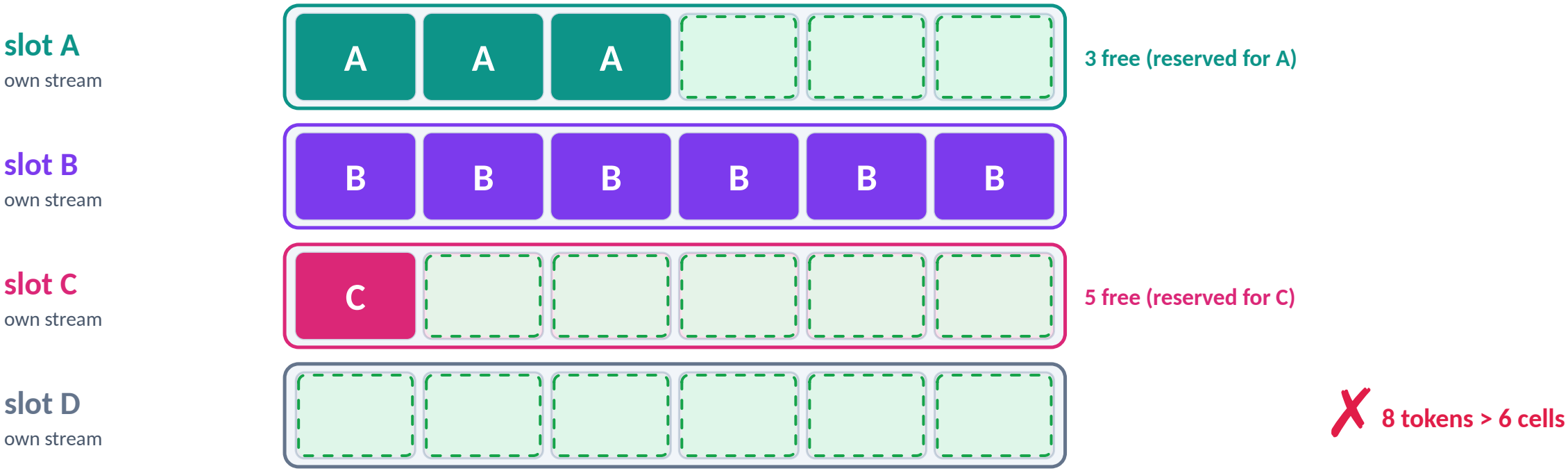


Each slot owns a fixed slab

No borrowing across slabs

Split KV: one slab per slot

Non-unified: every slot owns a fixed slab of cells



- **Each slot owns a fixed slab**
- **No borrowing across slabs**
- **14 of 24 cells free, yet D fails**

Unified KV: one shared pool of cells

Unified: all slots share one pool of cells



One pool, no slabs: any free cell works

Unified KV: one shared pool of cells

Unified: all slots share one pool of cells



One pool, no slabs: any free cell works

Decode = one token per slot per step

Unified KV: one shared pool of cells

Unified: all slots share one pool of cells



One pool, no slabs: any free cell works

Decode = one token per slot per step

D’s 8 tokens take 8 of the 14 free cells

Unified KV: one shared pool of cells

Unified: all slots share one pool of cells



One pool, no slabs: any free cell works

Decode = one token per slot per step

D's 8 tokens take 8 of the 14 free cells

Shared prompt: stored once, not four times

all four start with the same 3-token prompt

split: 4 copies = 12 cells

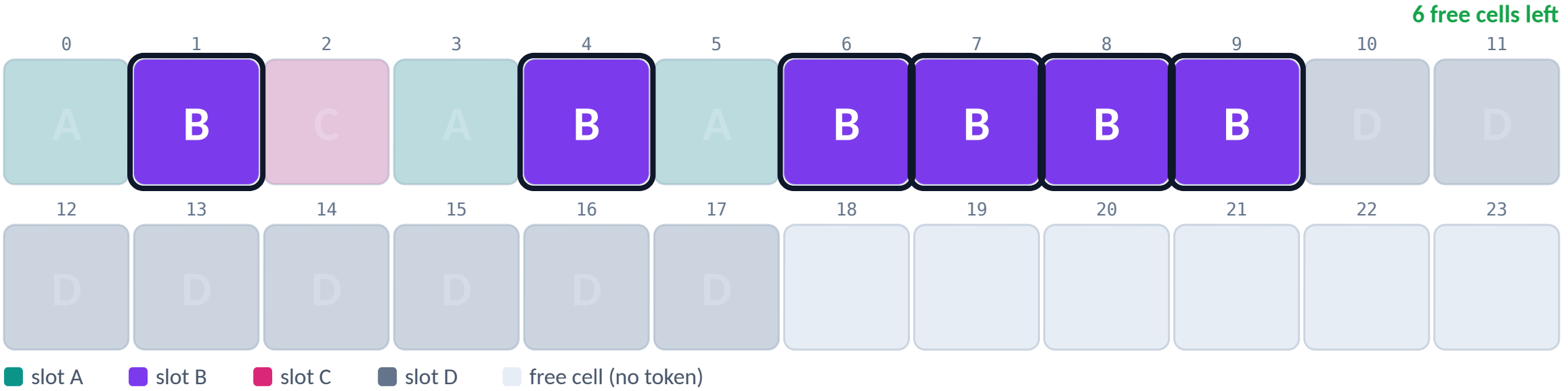


unified: 1 copy = 3 cells, 4 owners



Unified KV: one shared pool of cells

Unified: all slots share one pool of cells



One pool, no slabs: any free cell works

Decode = one token per slot per step

D’s 8 tokens take 8 of the 14 free cells

Shared prompt: stored once, not four times

Slot B is scattered: 3 pieces

all four start with the same 3-token prompt

split: 4 copies = 12 cells



unified: 1 copy = 3 cells, 4 owners



How attention uses a mask

Scores first, then the mask is added, then softmax.

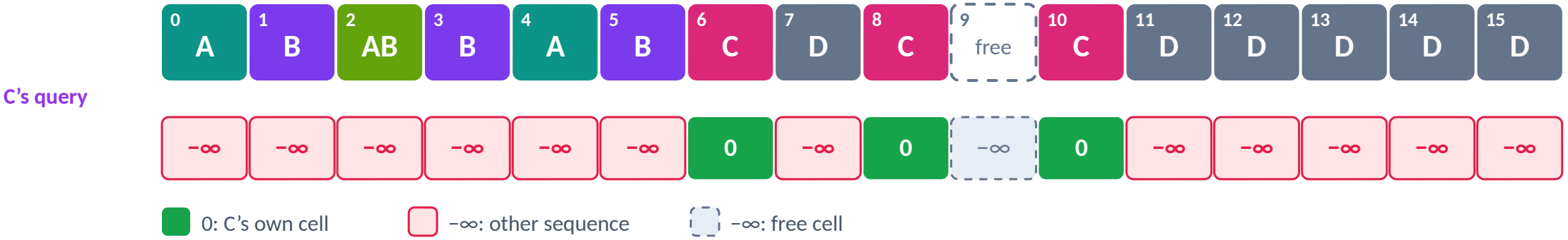


How attention uses a mask

Scores first, then the mask is added, then softmax.

The mask's main job in a unified pool: ownership

hide other sequences' cells, and free cells (no token there)



A query sees only its own sequence's cells

How attention uses a mask

Scores first, then the mask is added, then softmax.

C's query at cell 10 · window = 16 cells

	0 A	1 B	2 AB	3 B	4 A	5 B	6 C	7 D	8 C	9 free	10 C	11 D	12 D	13 D	14 D	15 D
score	0.8	2.8	3.1	1.2	2.4	0.9	2.1	1.7	0.4	1.1	1.3	3.0	0.6	2.2	1.5	1.9
+ mask	−∞	−∞	−∞	−∞	−∞	−∞	0	−∞	0	−∞	0	−∞	−∞	−∞	−∞	−∞
= masked	−∞	−∞	−∞	−∞	−∞	−∞	2.1	−∞	0.4	−∞	1.3	−∞	−∞	−∞	−∞	−∞
weight	0	0	0	0	0	0	0.61	0	0.11	0	0.28	0	0	0	0	0

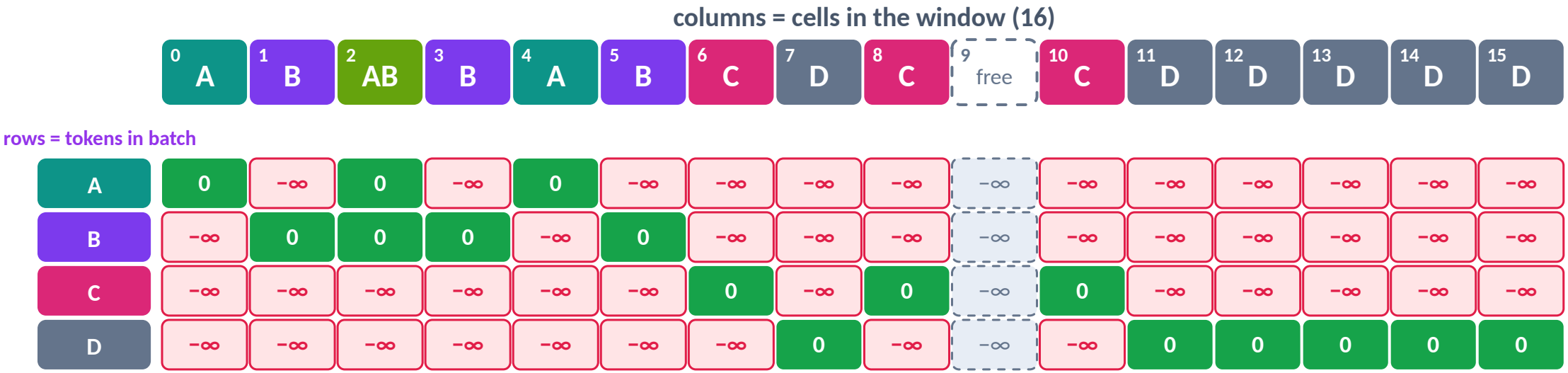
every cell gets a score first

add the mask, then softmax

weights only on C's cells · sum = 1.00

How attention uses a mask

Scores first, then the mask is added, then softmax.



4 tokens × 16 cells × 2 bytes = 128 bytes

2 bytes per entry (half precision)

built by the CPU, uploaded every step

the same mask is reused by every layer

How attention uses a mask

Scores first, then the mask is added, then softmax.

Split: one stream per sequence

C's stream holds only C's cells

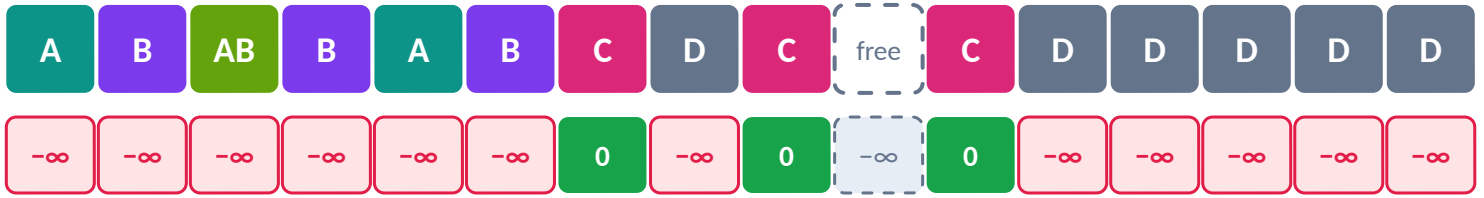


each row only as wide as C's own cells

(nothing from other sequences to hide)

Unified: one shared pool

C's query against everyone's cells



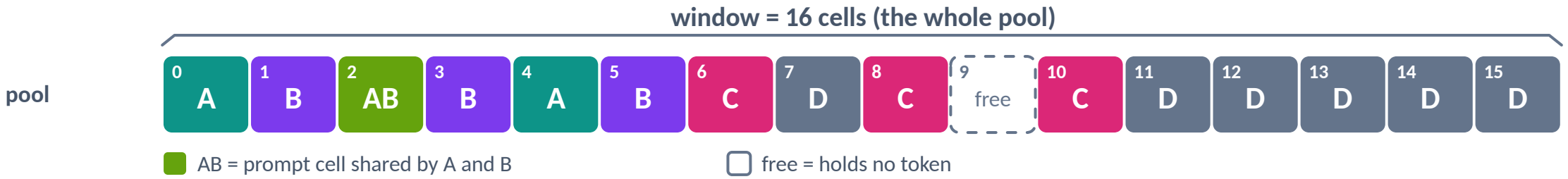
every query row spans the whole pool: 16 cells, 13 of them −∞

0 keeps the score other sequence free cell

Unified: one table over everyone's cells, so each row is as wide as the whole pool

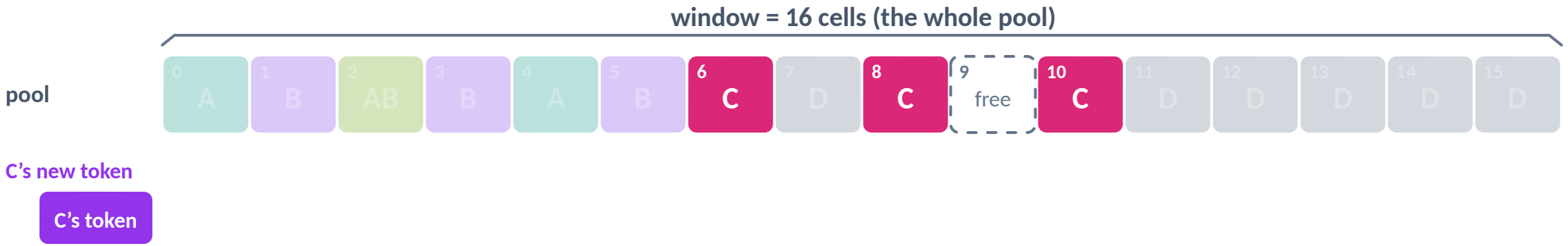
Every cell gets scored; the mask zeroes most weights

The kernel still loads them. C needs only its own cells.



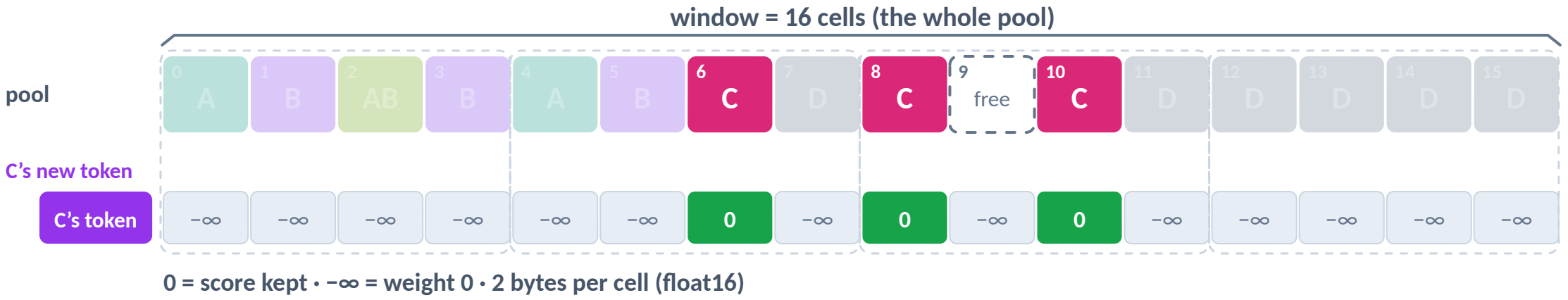
Every cell gets scored; the mask zeroes most weights

The kernel still loads them. C needs only its own cells.



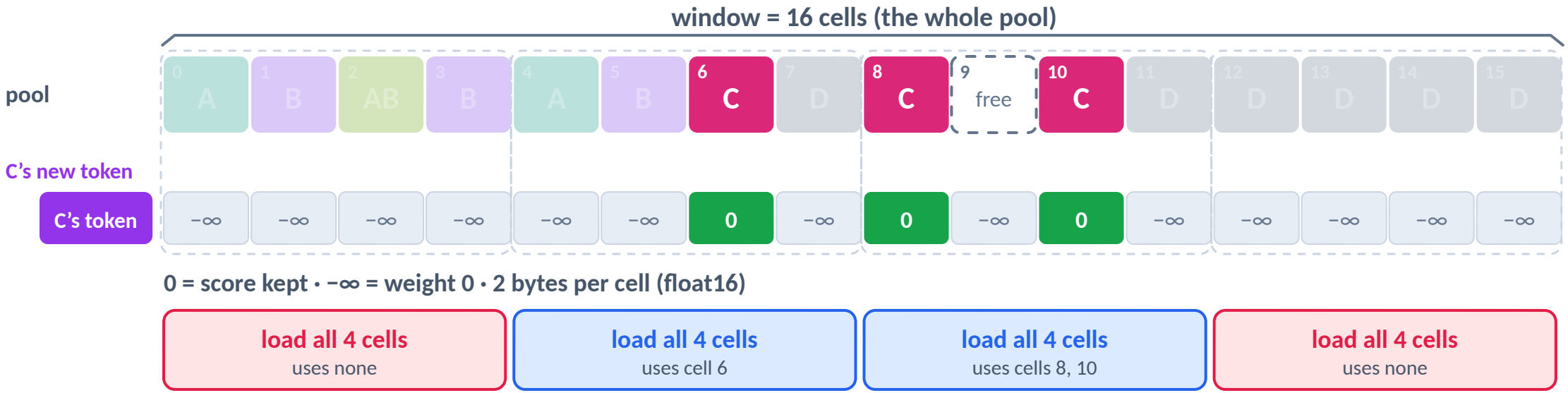
Every cell gets scored; the mask zeroes most weights

The kernel still loads them. C needs only its own cells.



Every cell gets scored; the mask zeroes most weights

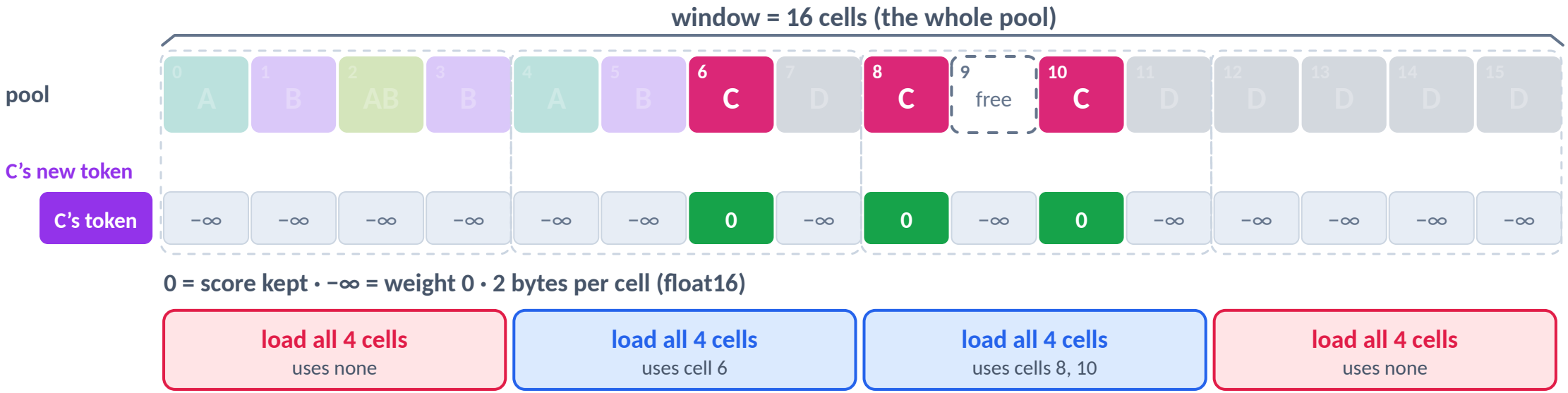
The kernel still loads them. C needs only its own cells.



tile = block of cells: 256 in reality, four here

Every cell gets scored; the mask zeroes most weights

The kernel still loads them. C needs only its own cells.



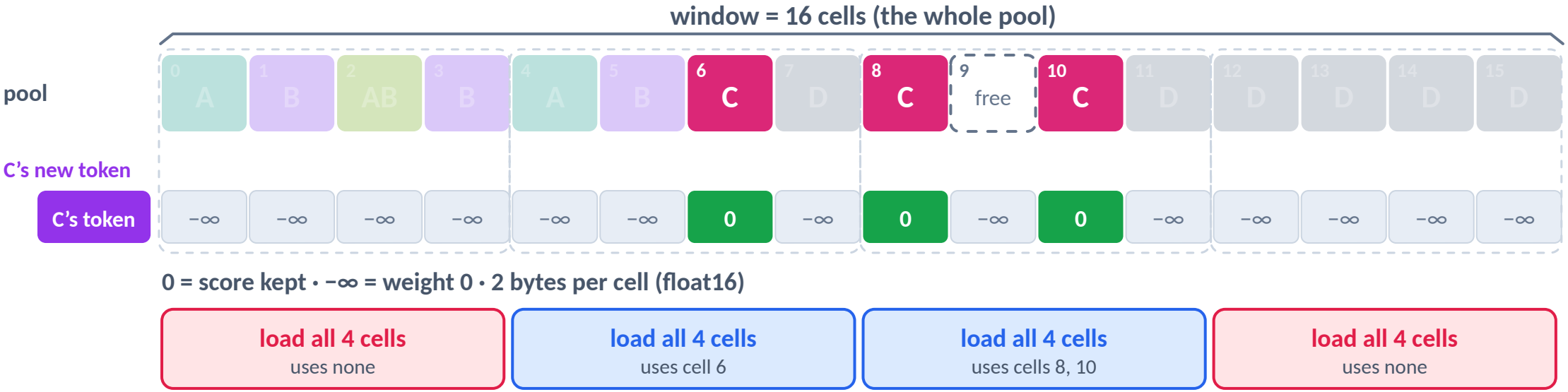
Most of the work is thrown away

16 cells loaded, 3 used

A trailing all-masked tile is skipped only in big batches; an interior one never.

Every cell gets scored; the mask zeroes most weights

The kernel still loads them. C needs only its own cells.



Most of the work is thrown away

16 cells loaded, 3 used

A trailing all-masked tile is skipped only in big batches; an interior one never.

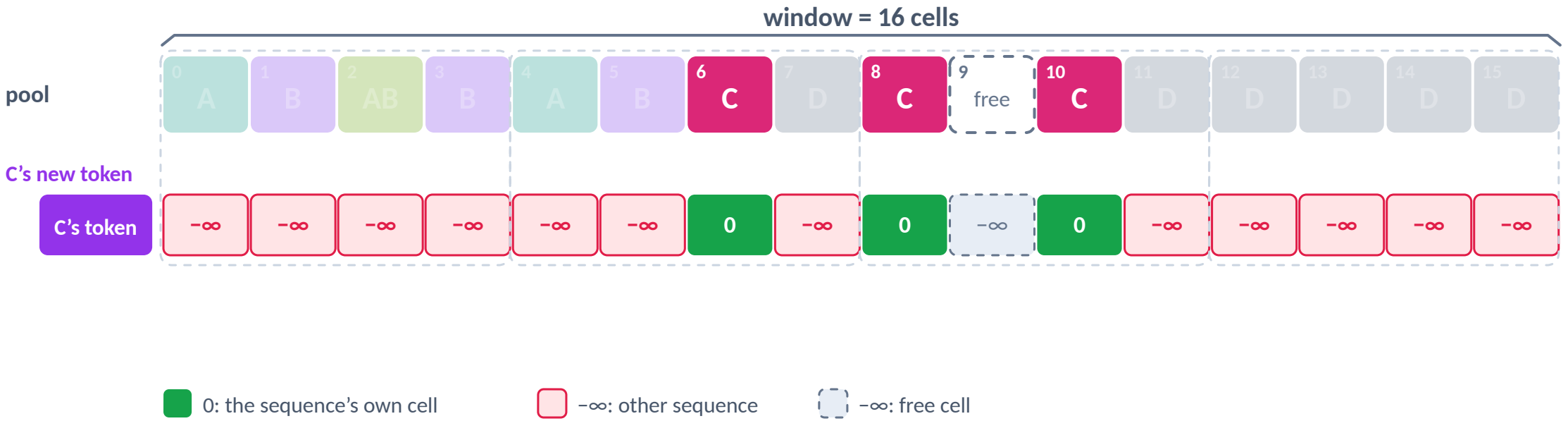
prefill speed · 8 sequences of 8K tokens

RTX 5090, Qwen3.8-27B



Approach 1: trim the window

Don’t read cells that can’t matter for this step.

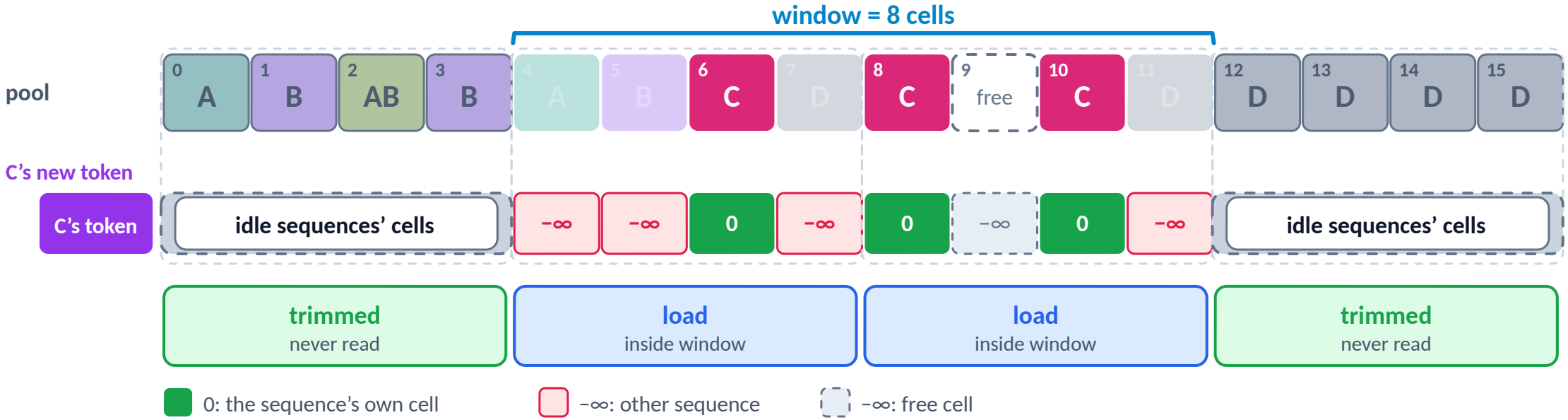


cells read per query

before 16 cells

Approach 1: trim the window

Don’t read cells that can’t matter for this step.

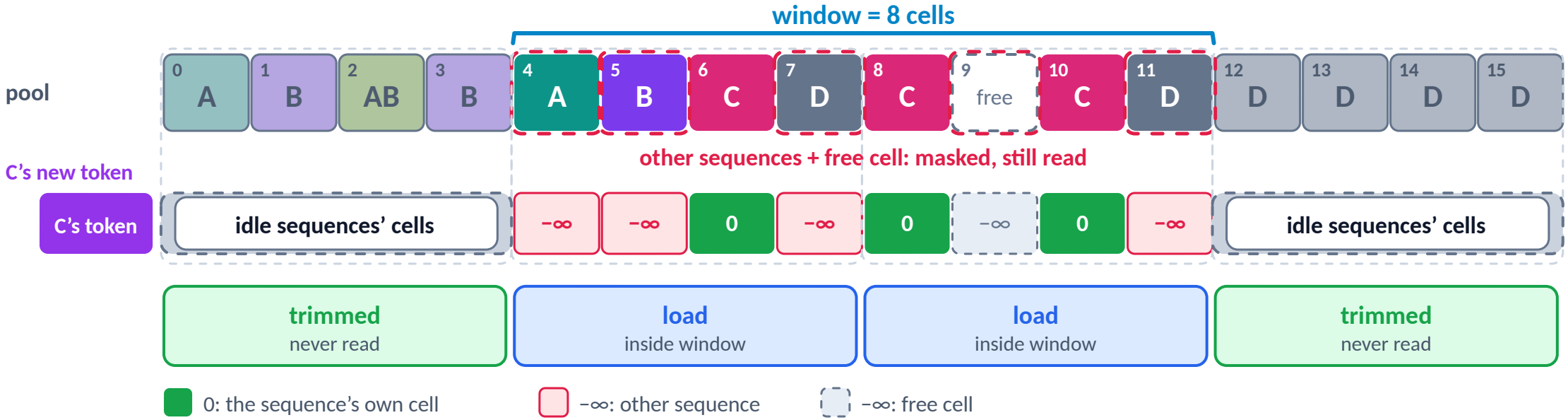


cells read per query



Approach 1: trim the window

Don’t read cells that can’t matter for this step.

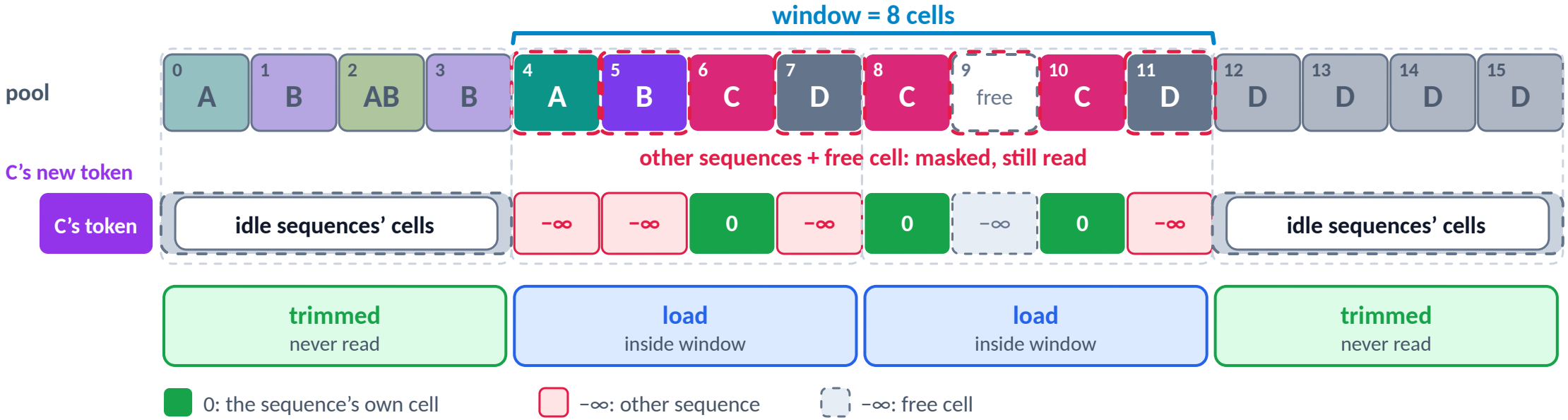


cells read per query

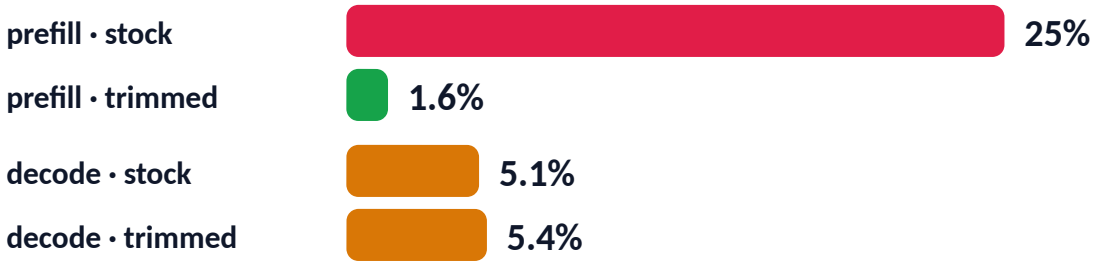


Approach 1: trim the window

Don’t read cells that can’t matter for this step.

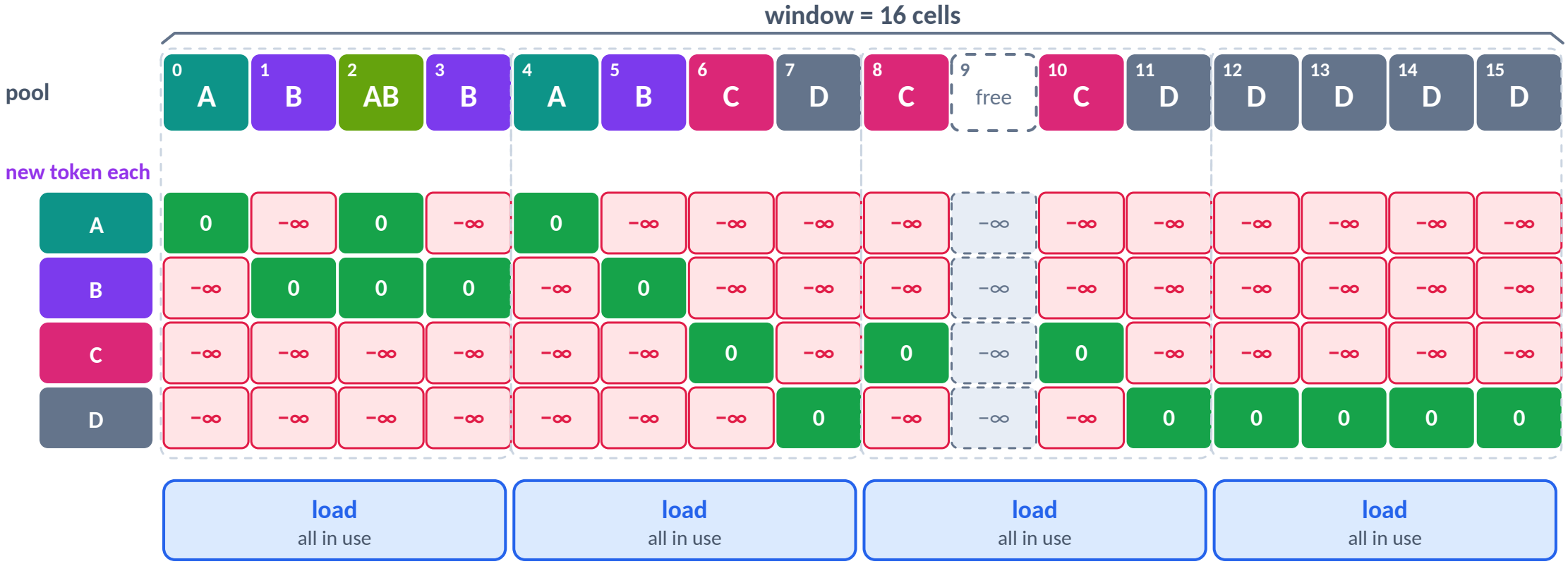






Approach 1: trim the window

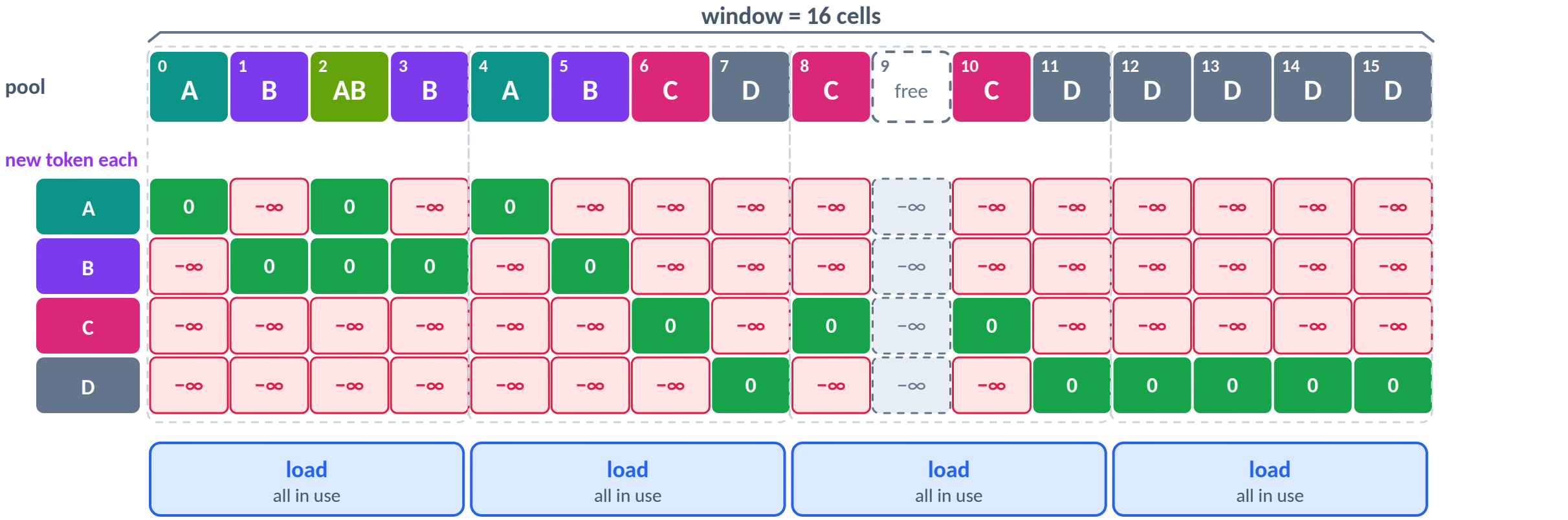
Don’t read cells that can’t matter for this step.



Many sequences: nothing to trim

Approach 1: trim the window

Don’t read cells that can’t matter for this step.



Many sequences: nothing to trim

Still a dense mask, sent every step

Approach 2: kv_rows, a list instead of a mask

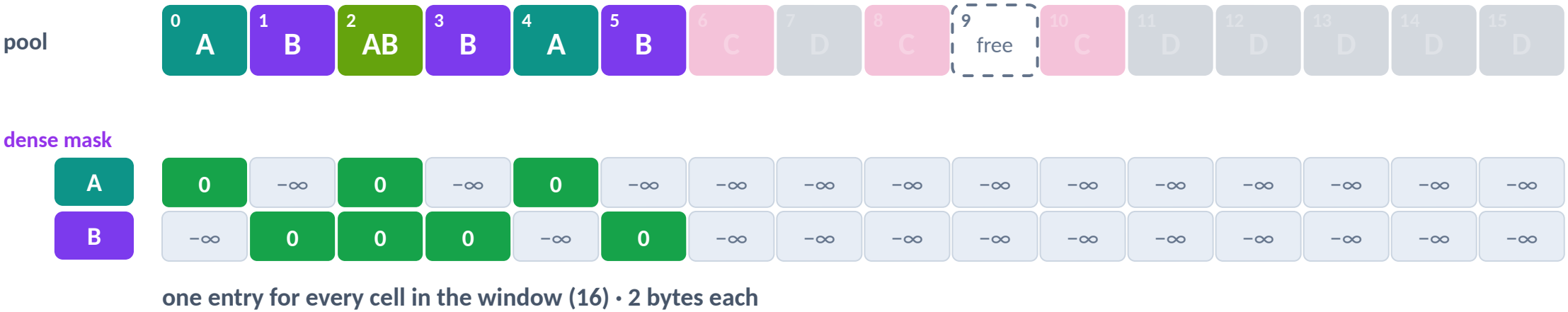
Send the cells to read, not a verdict on every cell.



A and B each decode one new token (C, D not running)

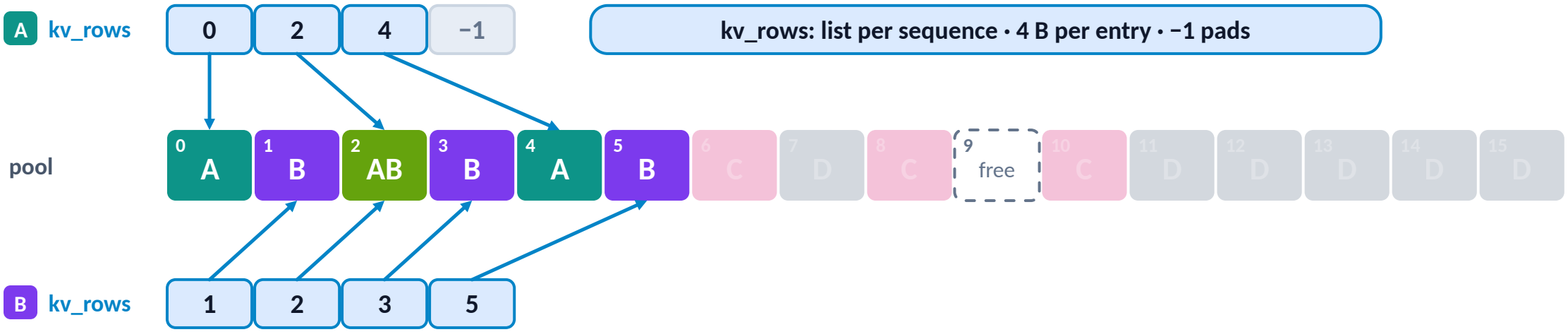
Approach 2: kv_rows, a list instead of a mask

Send the cells to read, not a verdict on every cell.



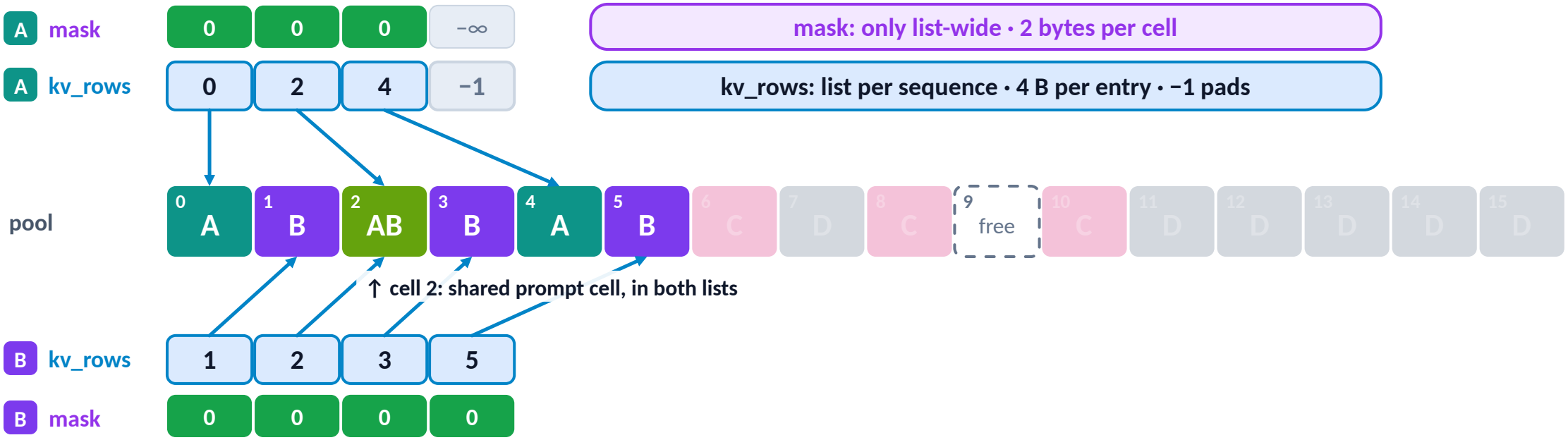
Approach 2: kv_rows, a list instead of a mask

Send the cells to read, not a verdict on every cell.



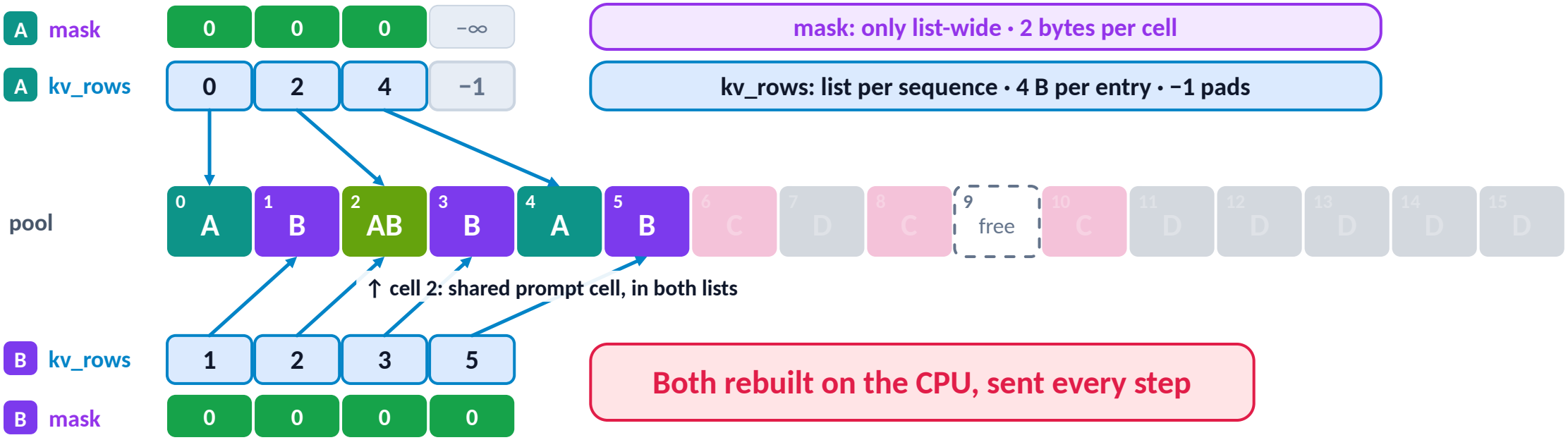
Approach 2: kv_rows, a list instead of a mask

Send the cells to read, not a verdict on every cell.



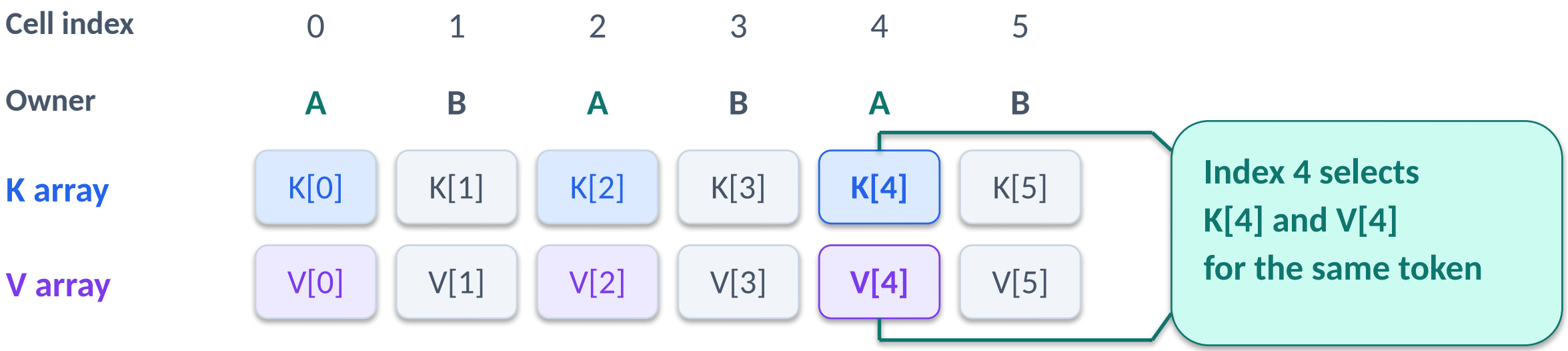
Approach 2: kv_rows, a list instead of a mask

Send the cells to read, not a verdict on every cell.



kv_rows: physical cell indices

One sequence owns many cells. The K and V arrays stay in place.



Sequence A: kv_rows = [0, 2, 4]

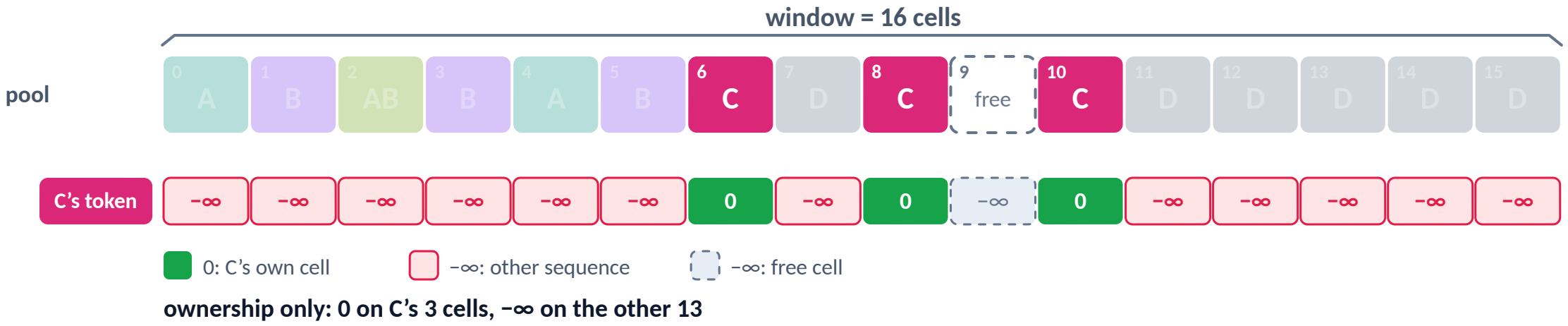
Sequence B: kv_rows = [1, 3, 5]

A row is a token vector, not a whole sequence.

Simplified view: one layer and one KV head; list padding omitted.

The mask with kv_rows: smaller and simpler

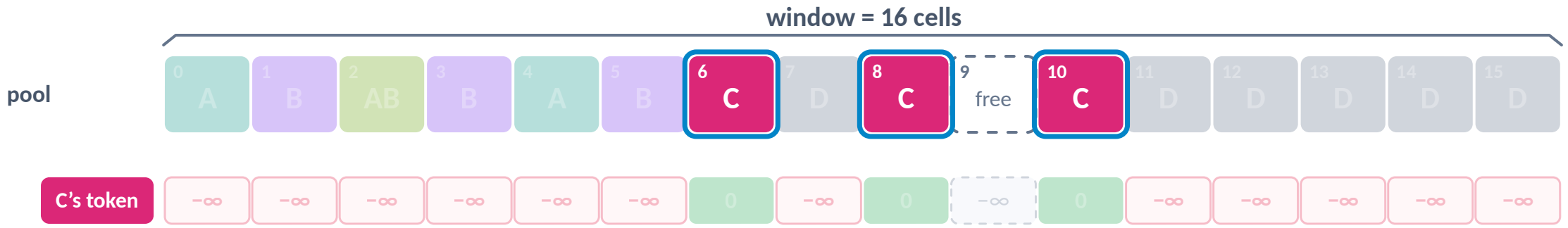
The list carries ownership; the mask keeps only what is left.



Cached tokens have no query: no attention, no causal mask for them

The mask with kv_rows: smaller and simpler

The list carries ownership; the mask keeps only what is left.



C kv_rows: 6 | 8 | 10 | −1

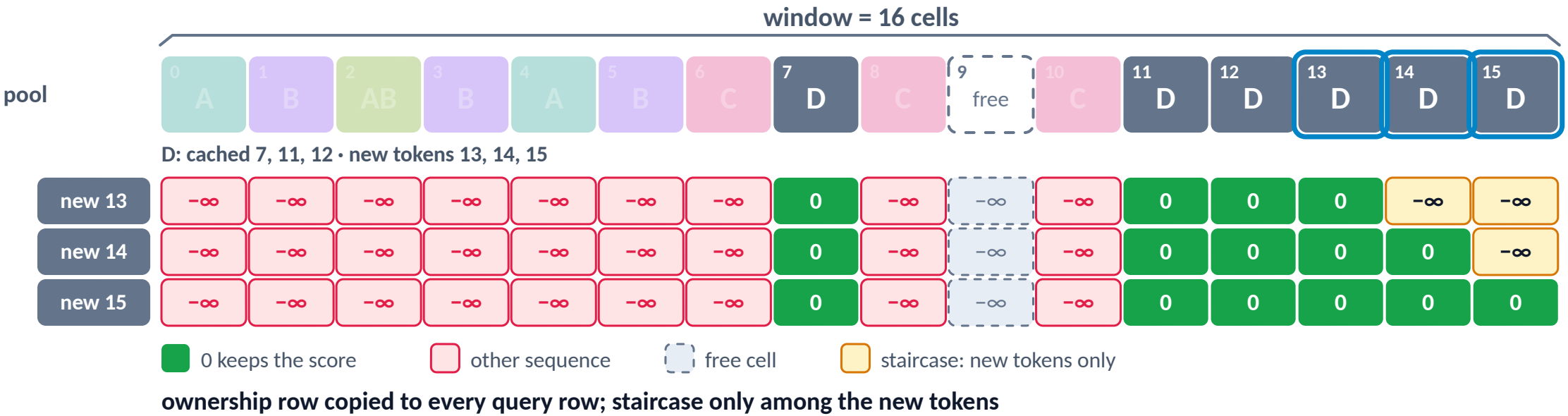
mask slice: 0 | 0 | 0 | −∞ end marker

Ownership moved into the list

the slice only marks the end: redundant in decode

The mask with kv_rows: smaller and simpler

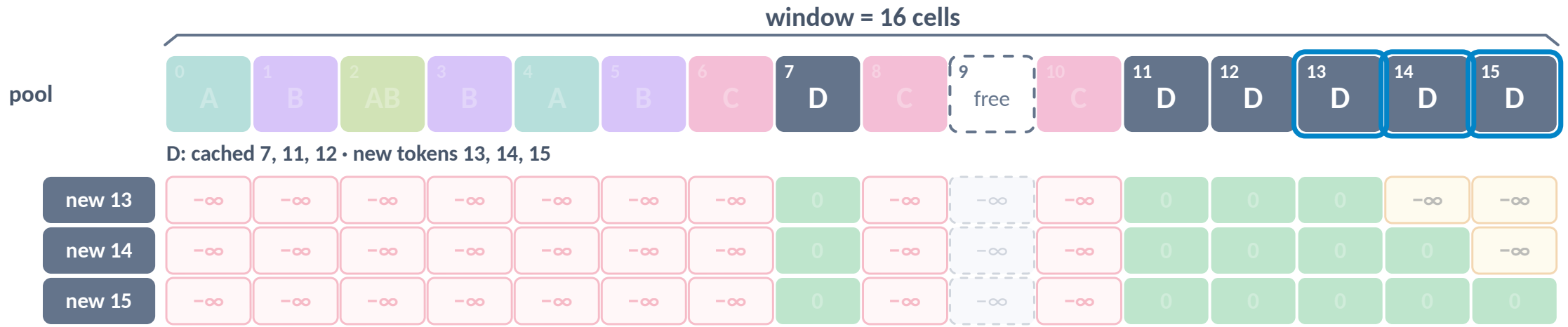
The list carries ownership; the mask keeps only what is left.



ownership row copied to every query row; staircase only among the new tokens

The mask with kv_rows: smaller and simpler

The list carries ownership; the mask keeps only what is left.

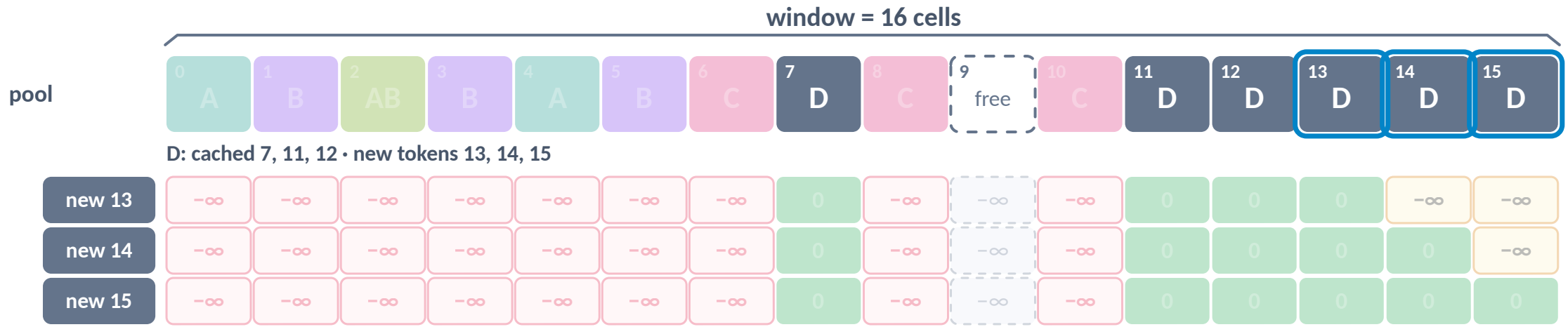


Cached columns all 0: no causal there

staircase only among the 3 new tokens

The mask with kv_rows: smaller and simpler

The list carries ownership; the mask keeps only what is left.



D kv_rows

	7	11	12	13	14	15
new 13	0	0	0	0	−∞	−∞
new 14	0	0	0	0	0	−∞
new 15	0	0	0	0	0	0

bytes per batch (2 B per entry)

toy · 3 queries

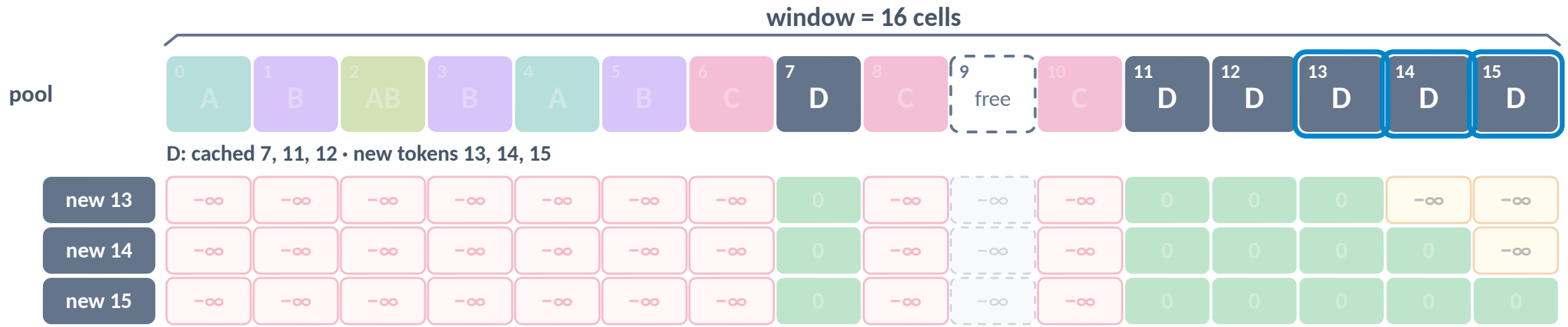
- dense: 16 × 3 = 96 B
- rows: 6 × 3 = 36 B (+ list)

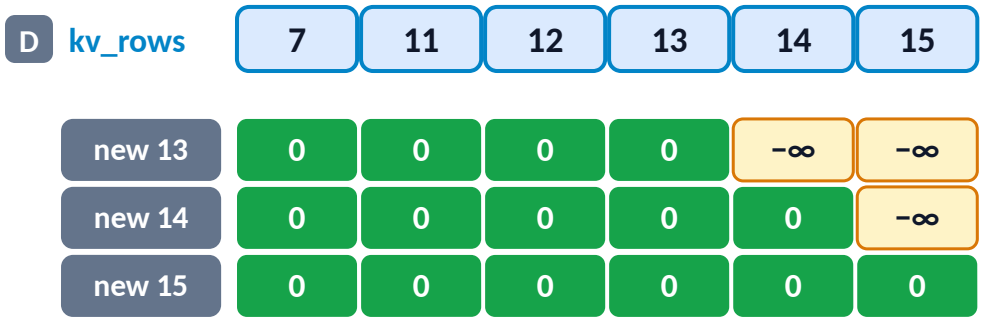
real · 512-token prefill

- dense: 124 MiB
- rows: 15.5 MiB — 8× smaller

The mask with kv_rows: smaller and simpler

The list carries ownership; the mask keeps only what is left.





one 2D mask table per sequence, like the split cache

kv_rows

- A · own width
- B · own width
- C · own width

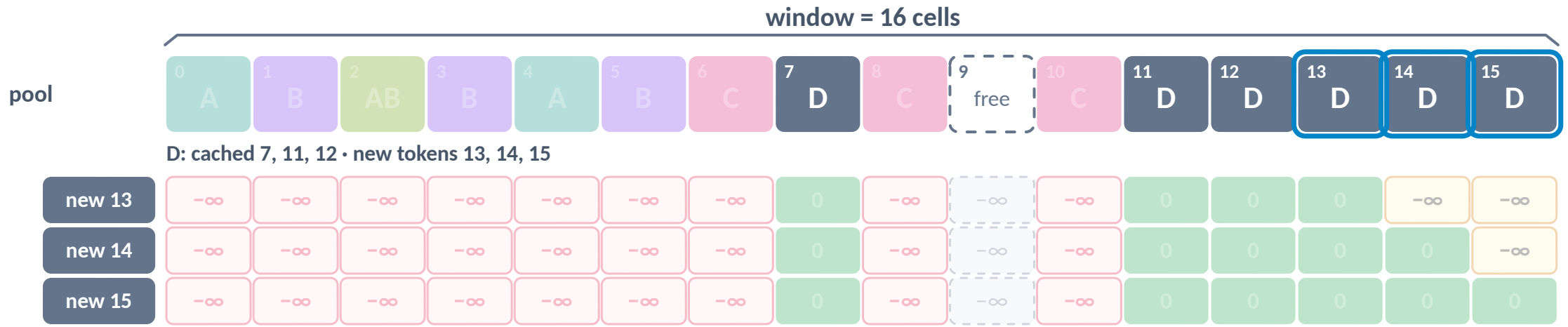
dense: one table over everyone's cells

each row as wide as the whole pool

(S × wider)

The mask with kv_rows: smaller and simpler

The list carries ownership; the mask keeps only what is left.

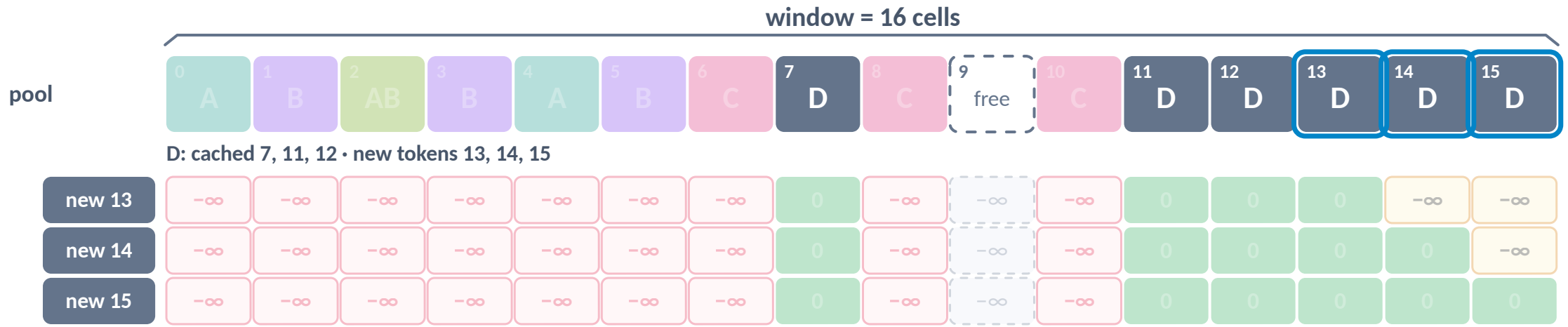


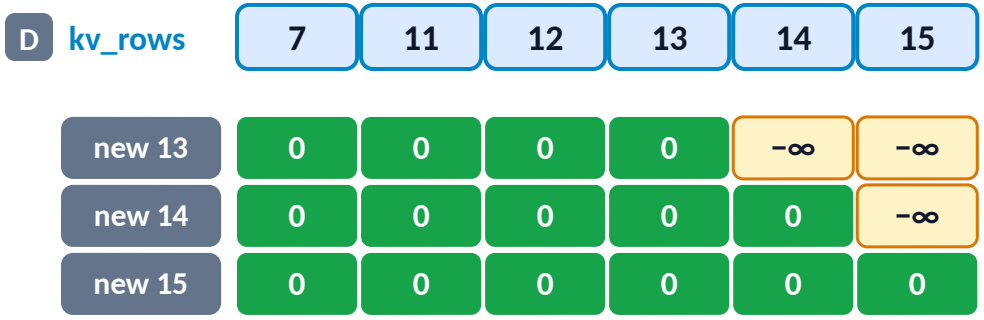
why it is still sent: the kernel contract

- the mask also carries sliding-window, ALiBi, non-causal cases
- padding: the kernel loads row 0 for −1, and the mask's −∞ cancels it
- always: softmax(score + mask)

The mask with kv_rows: smaller and simpler

The list carries ownership; the mask keeps only what is left.





I prototyped dropping it (own, not upstream)

the kernel builds causality from the list;
outputs identical (CUDA vs CPU tests, same output)
RTX 5090, 2B model, vs stock master:
decode about +4%, prefill +1.6 to +2.7%

A few percent: the mask upload was a small cost.

kv_rows: what it costs, what it buys

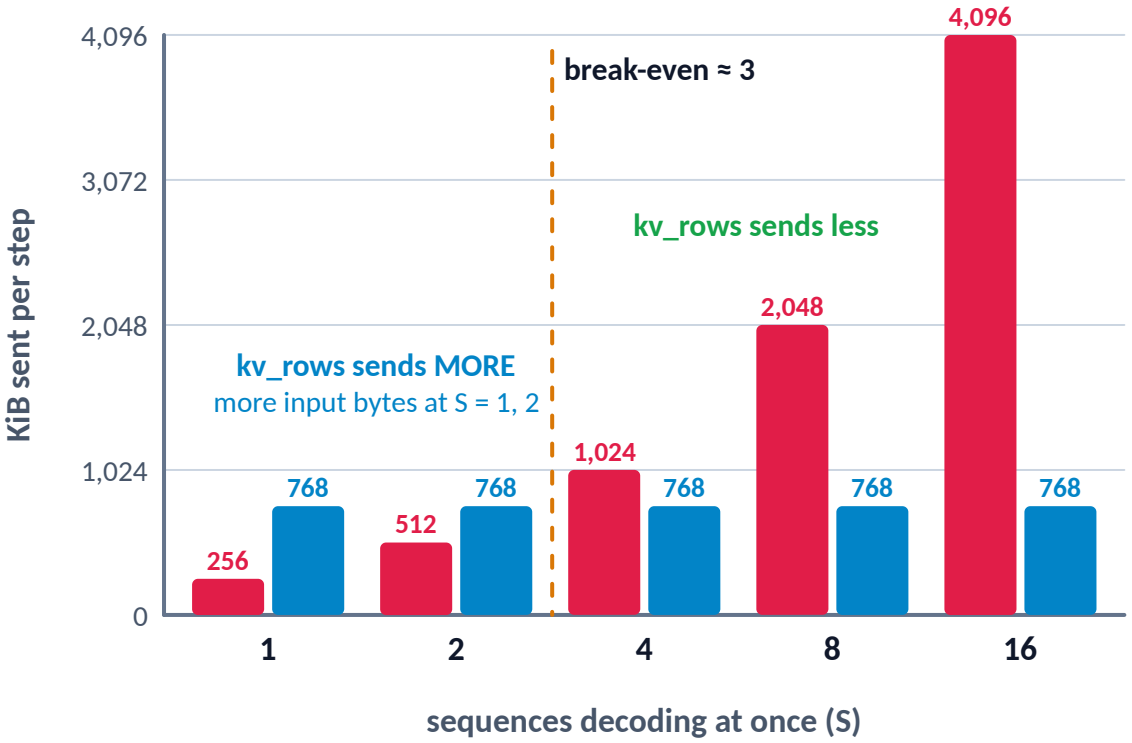
Small S: less work removed; list and gather costs remain.

Bytes sent per step · decode

one new token per sequence · derived, not measured

dense = 2 B × 128K × S **kv_rows** = (4+2) B × (128K/S) × S = 768 KiB

dense mask kv_rows list + mask



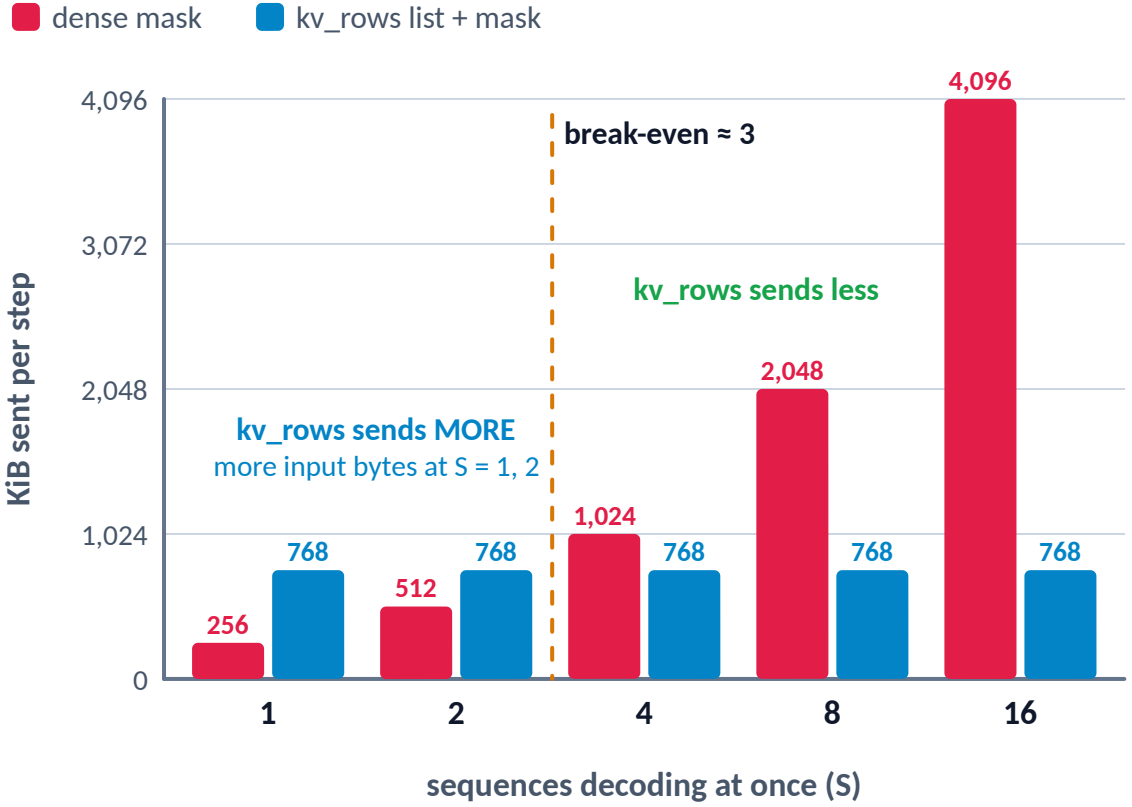
kv_rows: what it costs, what it buys

Small S: less work removed; list and gather costs remain.

Bytes sent per step · decode

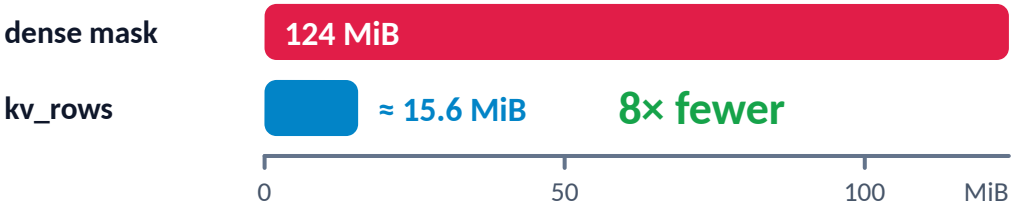
one new token per sequence · derived, not measured

dense = 2 B × 128K × S **kv_rows** = (4+2) B × (128K/S) × S = 768 KiB



Bytes sent · one 512-token prefill batch

8 sequences × 15,872 = 126,976 cells · derived



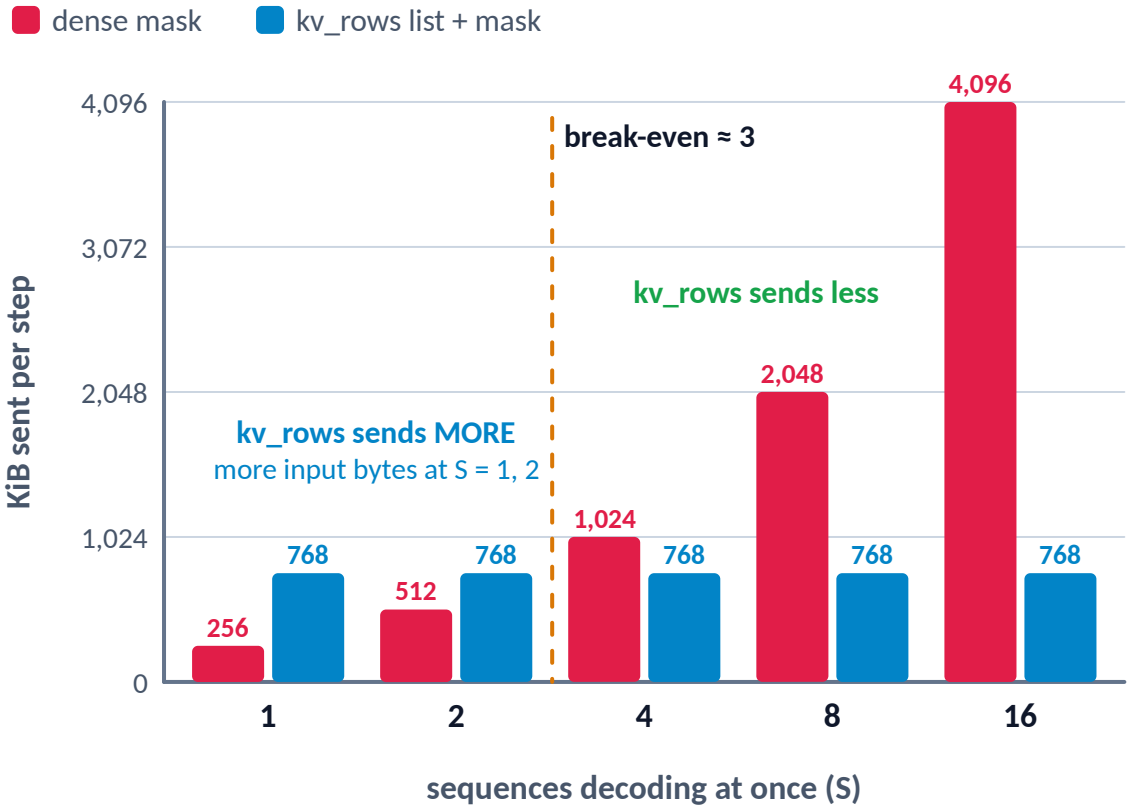
kv_rows: what it costs, what it buys

Small S: less work removed; list and gather costs remain.

Bytes sent per step · decode

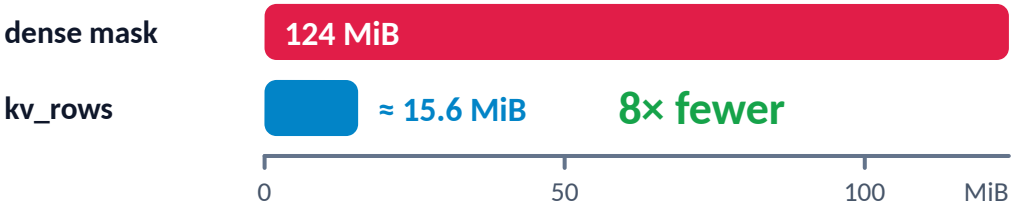
one new token per sequence · derived, not measured

dense = 2 B × 128K × S **kv_rows** = (4+2) B × (128K/S) × S = 768 KiB



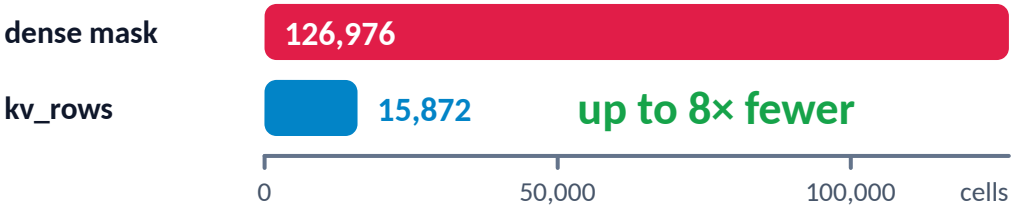
Bytes sent · one 512-token prefill batch

8 sequences × 15,872 = 126,976 cells · derived



Cells visited per query token

same prefill · one sequence in the 8-sequence pool



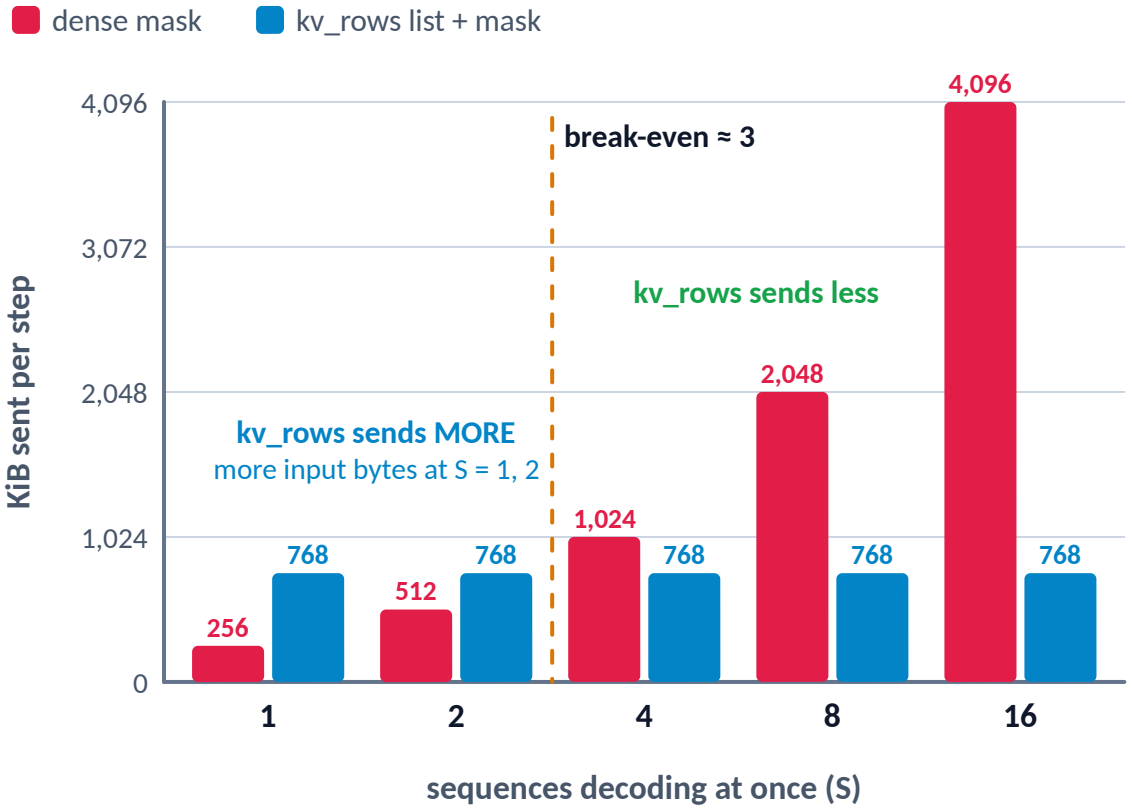
kv_rows: what it costs, what it buys

Small S: less work removed; list and gather costs remain.

Bytes sent per step · decode

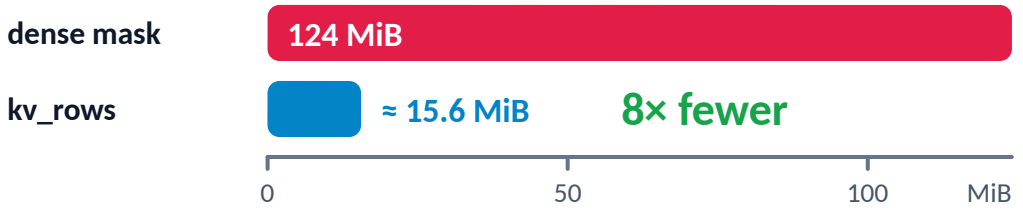
one new token per sequence · derived, not measured

dense = 2 B × 128K × S **kv_rows** = (4+2) B × (128K/S) × S = 768 KiB



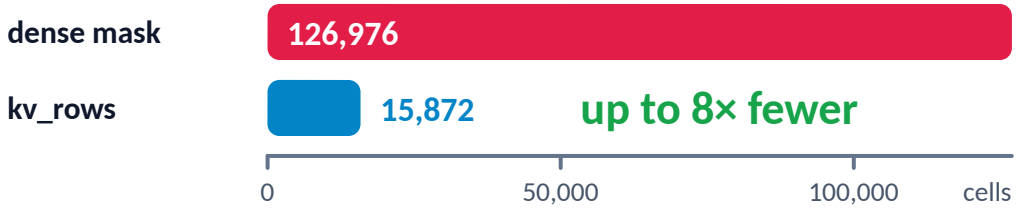
Bytes sent · one 512-token prefill batch

8 sequences × 15,872 = 126,976 cells · derived



Cells visited per query token

same prefill · one sequence in the 8-sequence pool



Low S: less cross-sequence work removed

Prefill: one sequence, up to 8× less to read

kv_rows: what it costs, what it buys

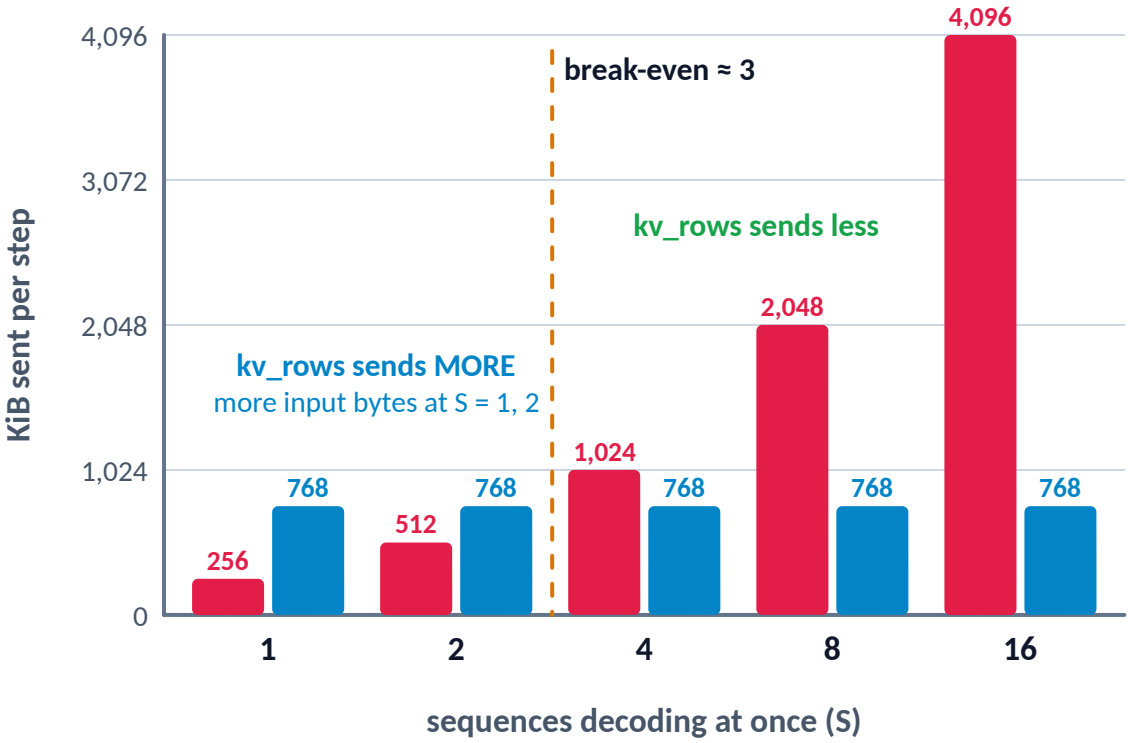
Small S: less work removed; list and gather costs remain.

Bytes sent per step · decode

one new token per sequence · derived, not measured

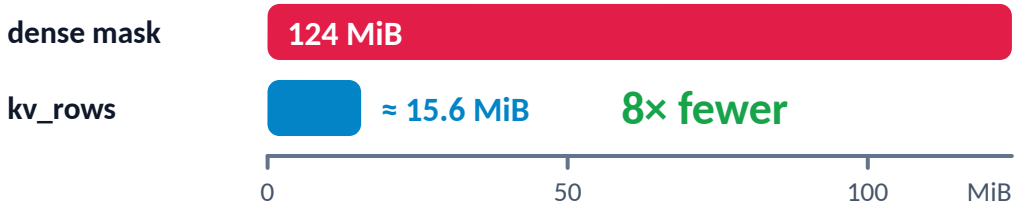
dense = 2 B × 128K × S **kv_rows** = (4+2) B × (128K/S) × S = 768 KiB

dense mask kv_rows list + mask



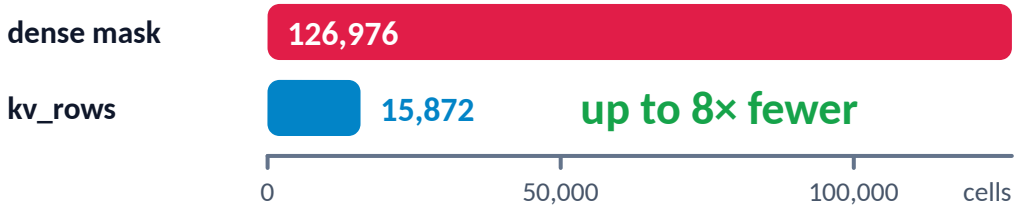
Bytes sent · one 512-token prefill batch

8 sequences × 15,872 = 126,976 cells · derived



Cells visited per query token

same prefill · one sequence in the 8-sequence pool



Prefill 2.5–2.9× faster

Aman's change · RTX PRO 6000 · 4 sequences × ~100K tokens

Prefill: one sequence, up to 8× less to read

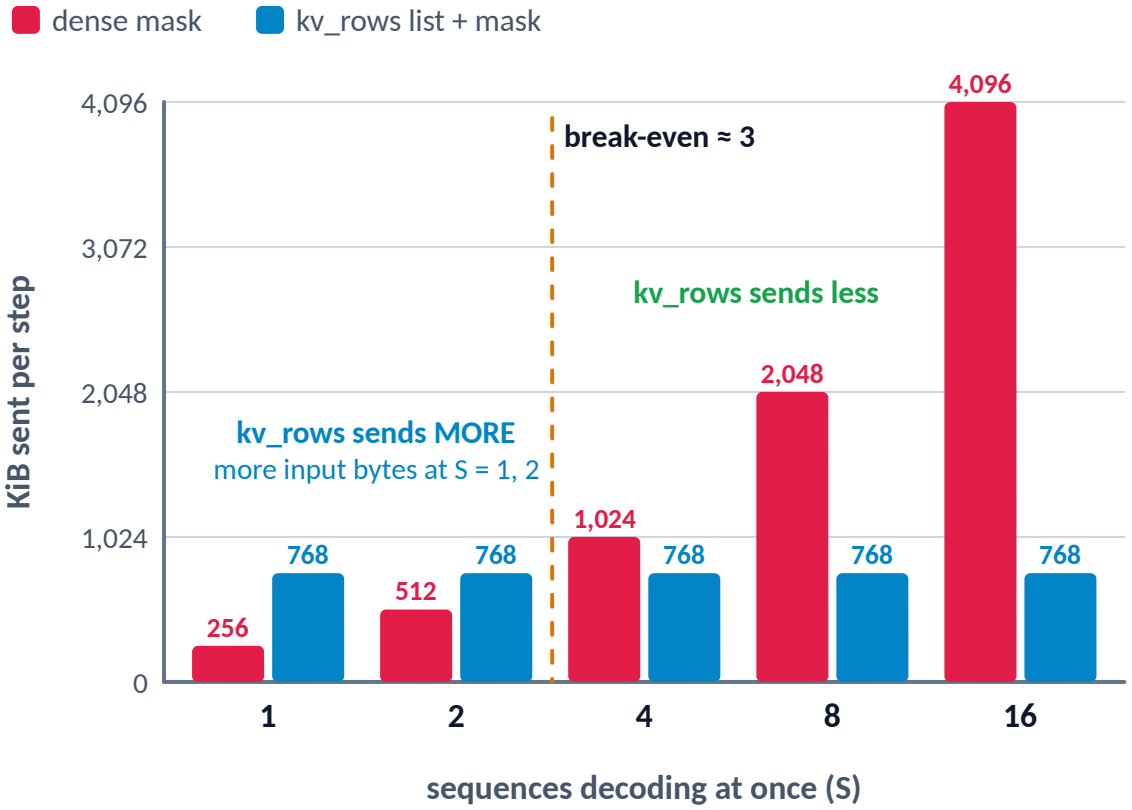
kv_rows: what it costs, what it buys

Small S: less work removed; list and gather costs remain.

Bytes sent per step · decode

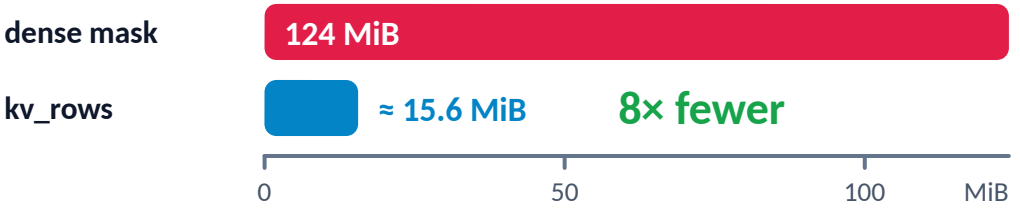
one new token per sequence · derived, not measured

dense = 2 B × 128K × S **kv_rows** = (4+2) B × (128K/S) × S = 768 KiB



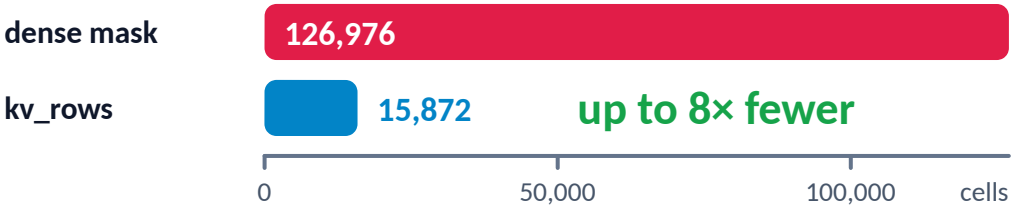
Bytes sent · one 512-token prefill batch

8 sequences × 15,872 = 126,976 cells · derived



Cells visited per query token

same prefill · one sequence in the 8-sequence pool



Prefill 2.5–2.9× faster

Aman's change · RTX PRO 6000 · 4 sequences × ~100K tokens

One sequence: nothing to skip

still pay to build the list and gather (Georgi's objection)

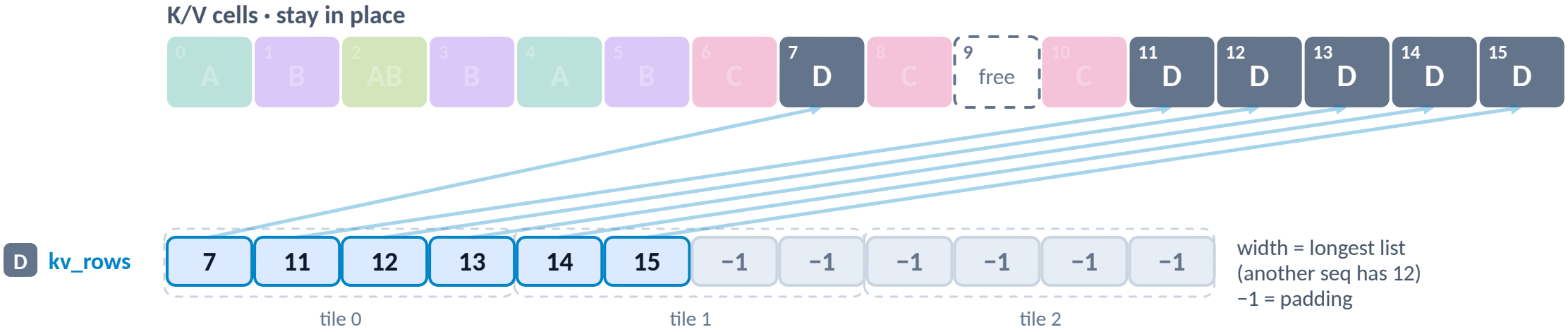
Inside the kernel: gather by index

The tile loop runs over the list, not over the window.



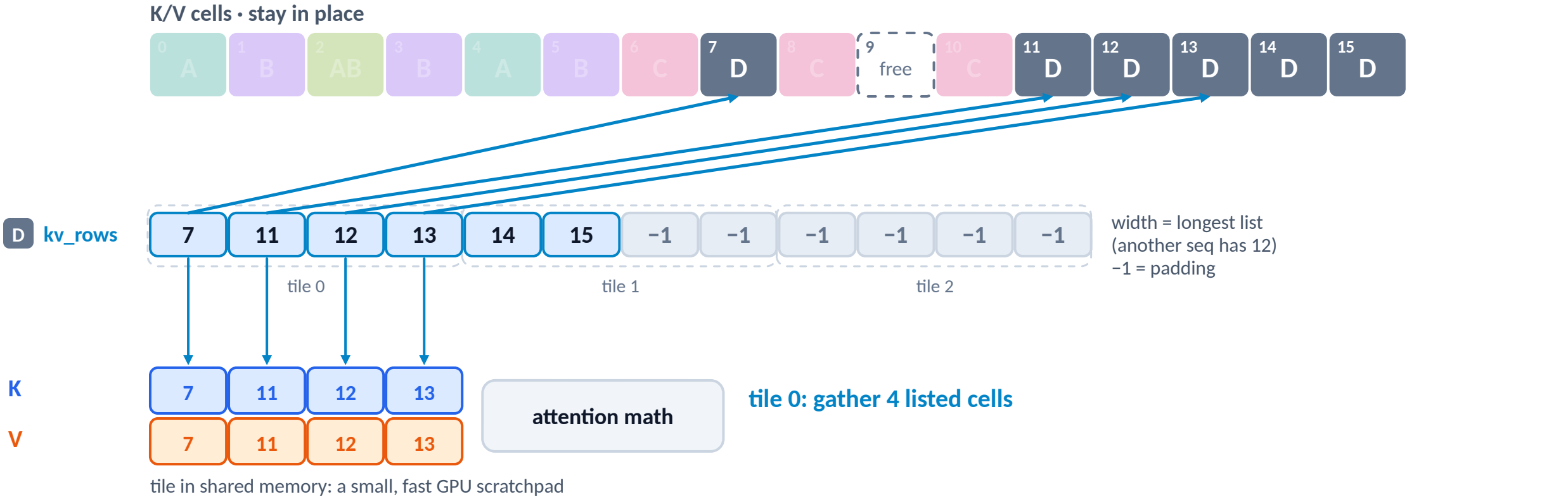
Inside the kernel: gather by index

The tile loop runs over the list, not over the window.



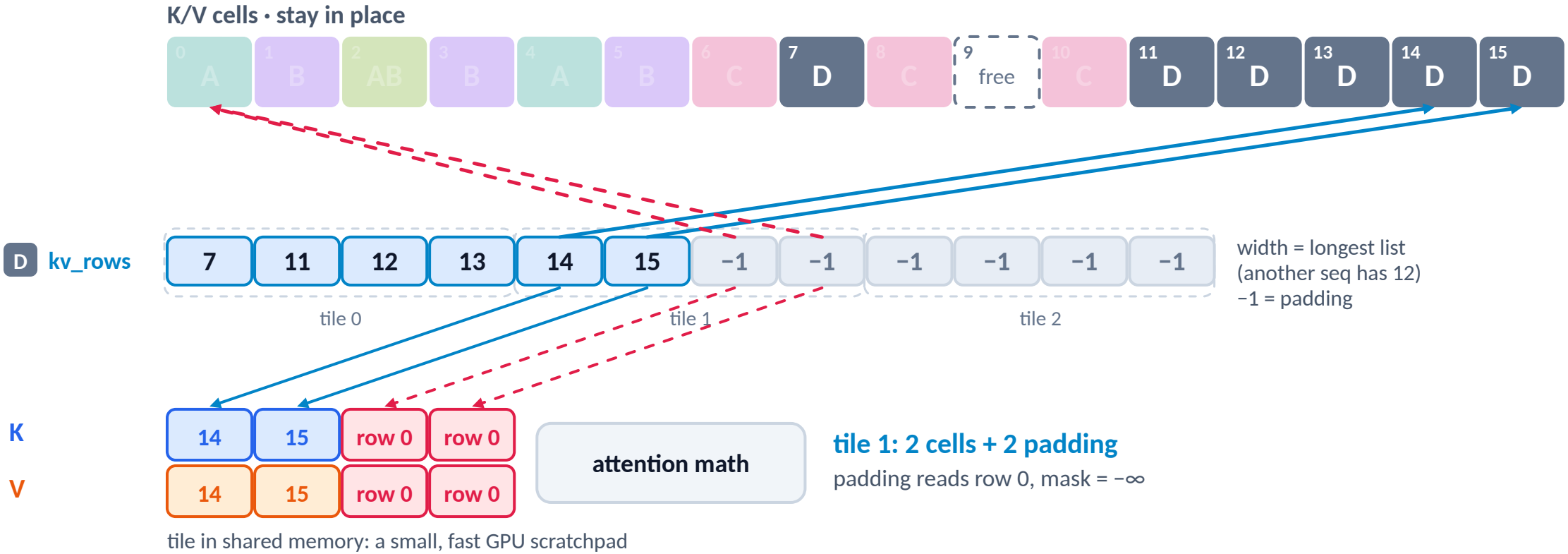
Inside the kernel: gather by index

The tile loop runs over the list, not over the window.



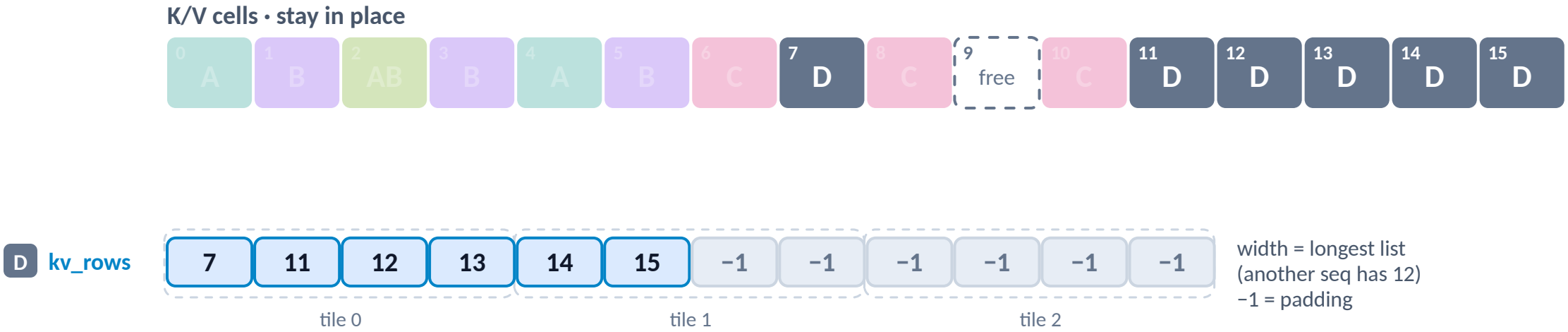
Inside the kernel: gather by index

The tile loop runs over the list, not over the window.



Inside the kernel: gather by index

The tile loop runs over the list, not over the window.



Inside the kernel: gather by index

The tile loop runs over the list, not over the window.

K/V cells · stay in place

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	AB	B	A	B	C	D	C	free	C	D	D	D	D	D

D kv_rows

7	11	12	13	14	15	−1	−1	−1	−1	−1	−1
tile 0				tile 1				tile 2			

width = longest list
(another seq has 12)
−1 = padding

K

V

attention math

tile 2: all padding, skipped
no loads, no math

tile in shared memory: a small, fast GPU scratchpad

dense

wasted	load	load	load
uses 0 of 4	uses 1 of 4	uses 1 of 4	uses 4 of 4

16 loaded
6 used

kv_rows

load	load	skipped
uses 4 of 4	uses 2 of 4	all padding

8 loaded, 6 used

Inside the kernel: gather by index

The tile loop runs over the list, not over the window.

K/V cells · stay in place

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	AB	B	A	B	C	D	C	free	C	D	D	D	D	D

D kv_rows

7	11	12	13	14	15	−1	−1	−1	−1	−1	−1
tile 0				tile 1				tile 2			

width = longest list (another seq has 12)
−1 = padding

K

V

attention math

tile 2: all padding, skipped
no loads, no math

tile in shared memory: a small, fast GPU scratchpad

dense

wasted	load	load	load
uses 0 of 4	uses 1 of 4	uses 1 of 4	uses 4 of 4

16 loaded
6 used

kv_rows

load	load	skipped
uses 4 of 4	uses 2 of 4	all padding

8 loaded, 6 used

Half the loads