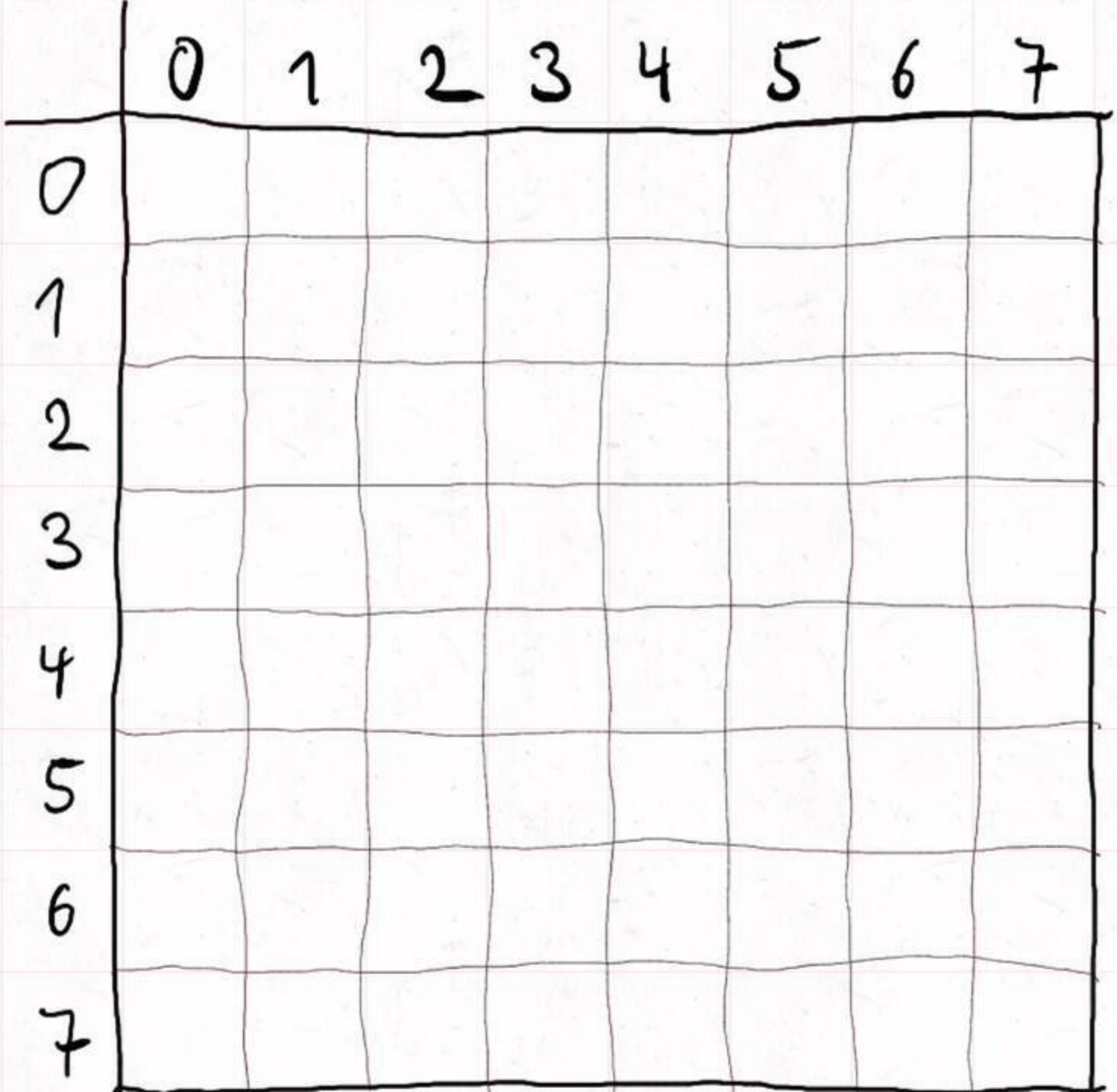
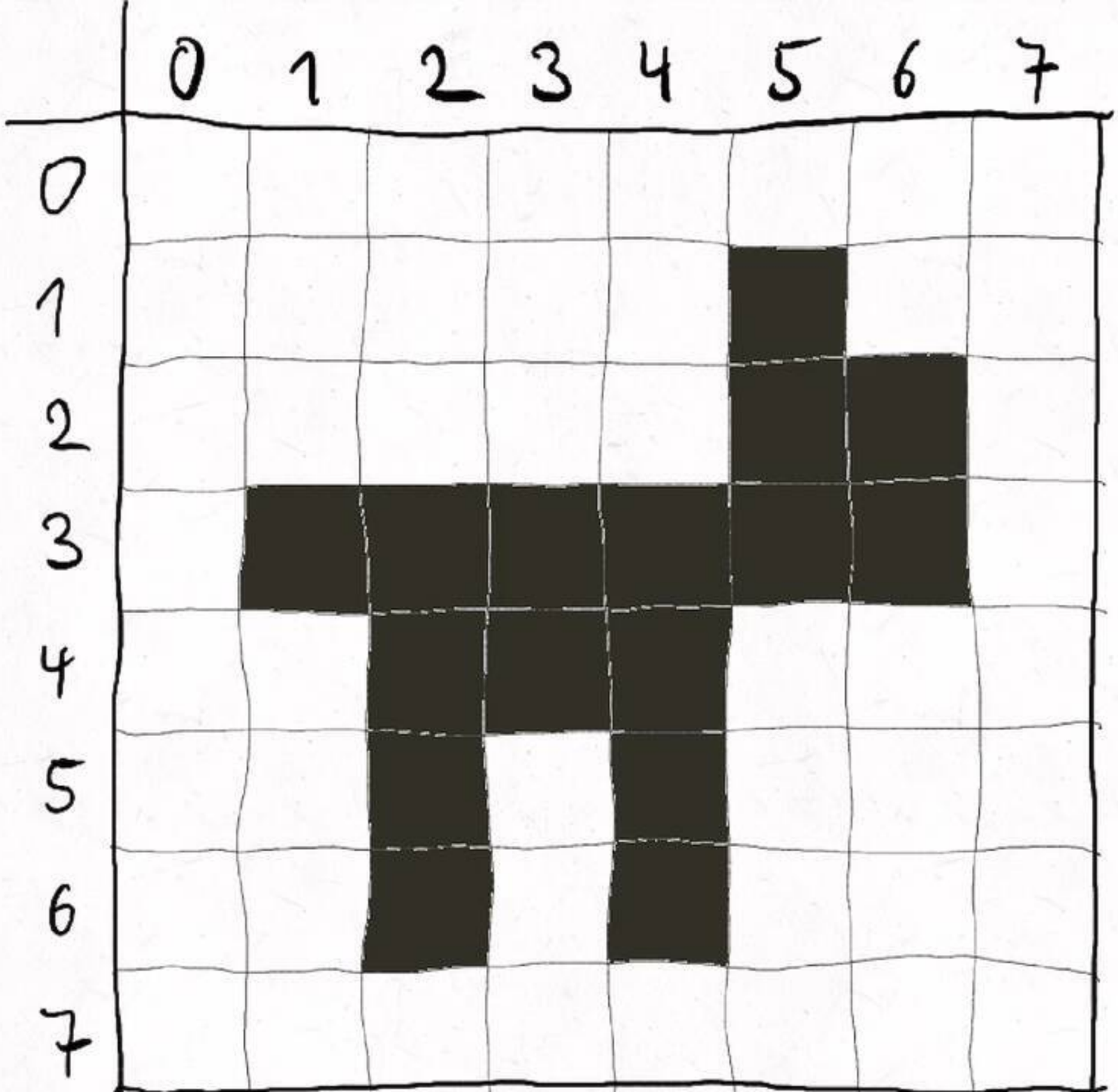
Work Partioning

Aufteilung einer Berechnung für die parallele Verarbeitung

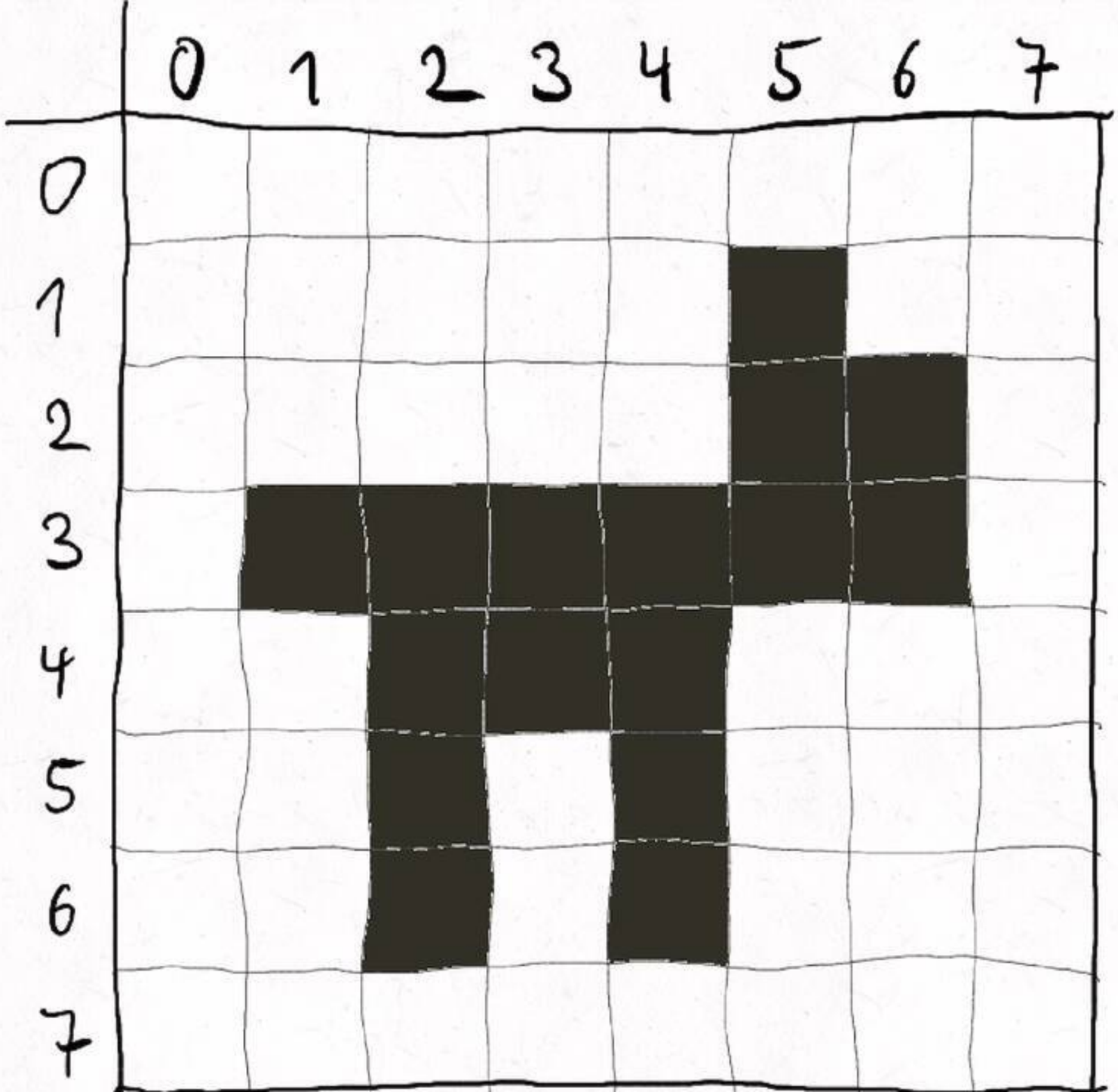
Sebastian Speiser



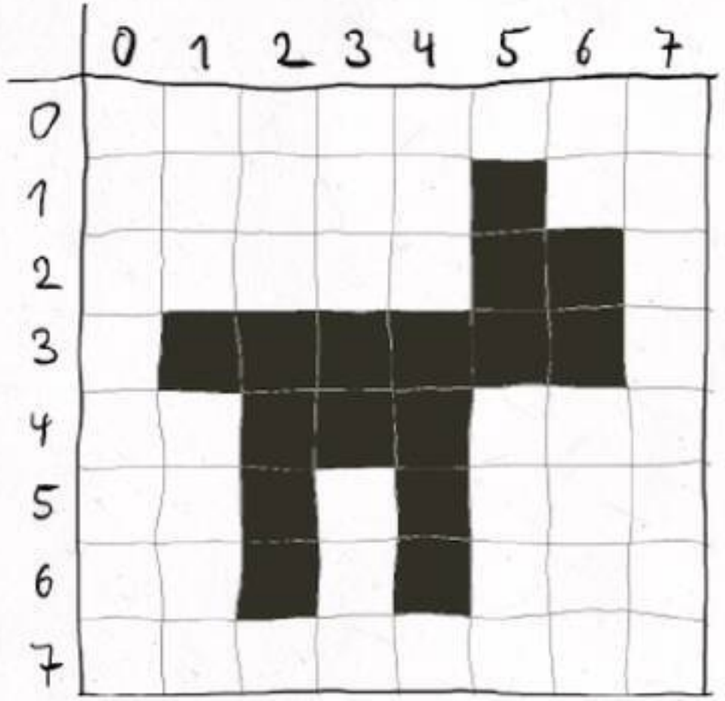
Gegeben eine Matrix

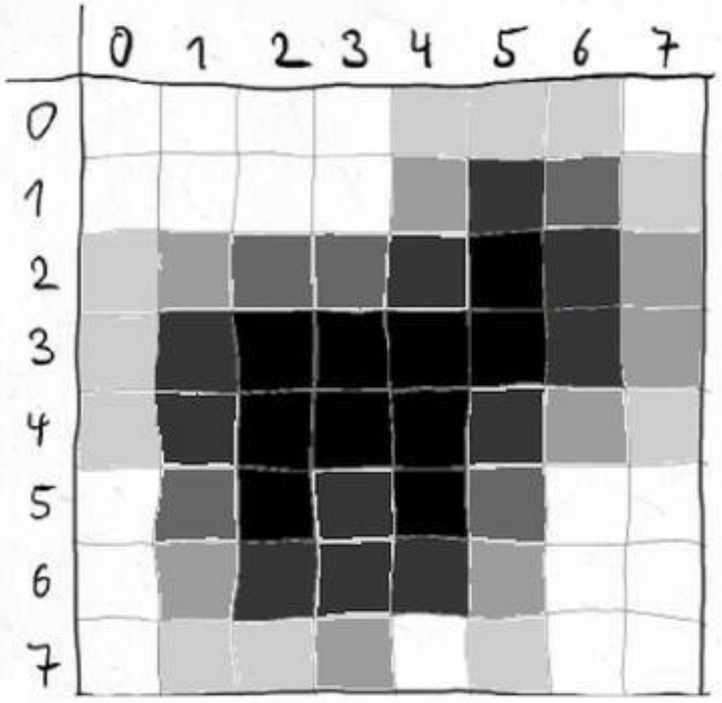


Gegeben eine Matrix, z.B. ein Bild



Gegeben eine Matrix, z.B. ein Bild (von einem Hund!?)





Gegeben eine Matrix, z.B. ein Bild (von einem Hund!?)

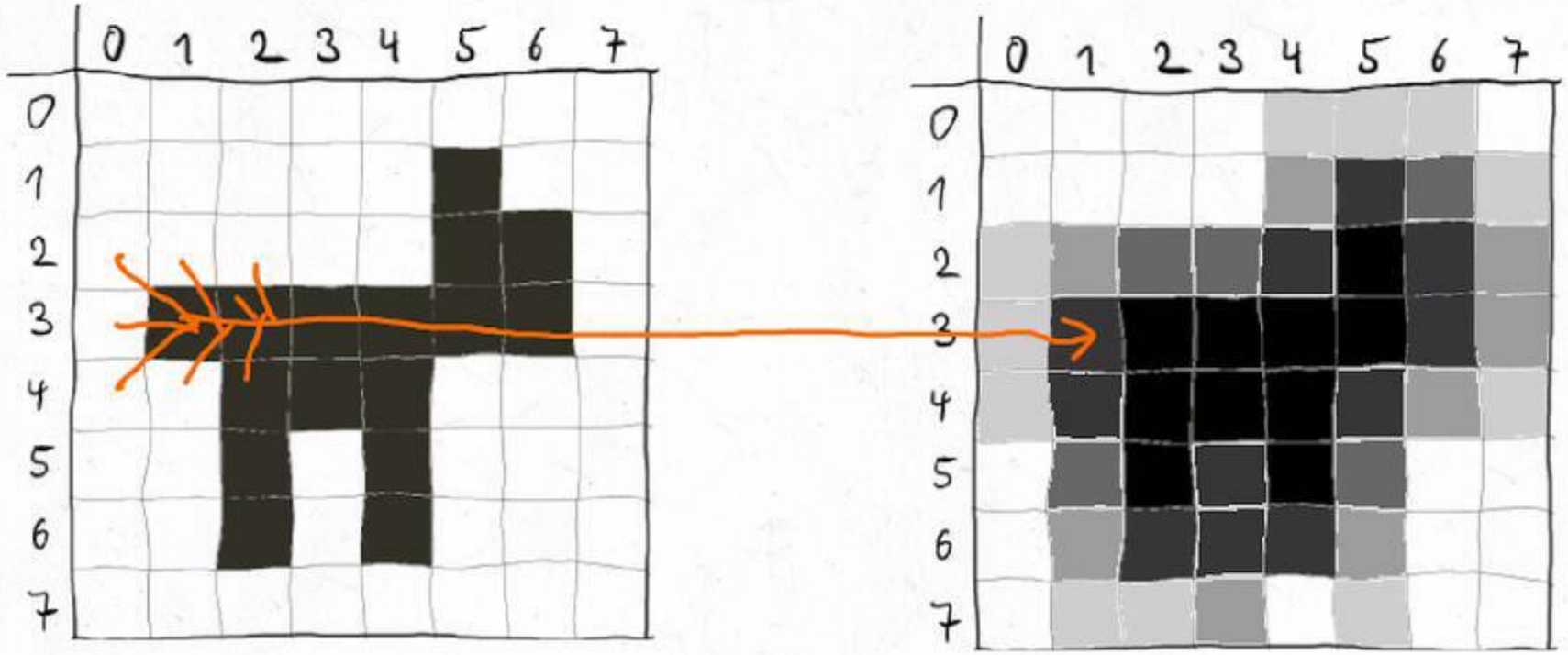
Es wird eine Schablone angewandt, d.h. jeder Wert (Pixel) im Ziel ist eine gewichtete Summe der umgebenden Pixel

↳ D. Schablone, z.B.

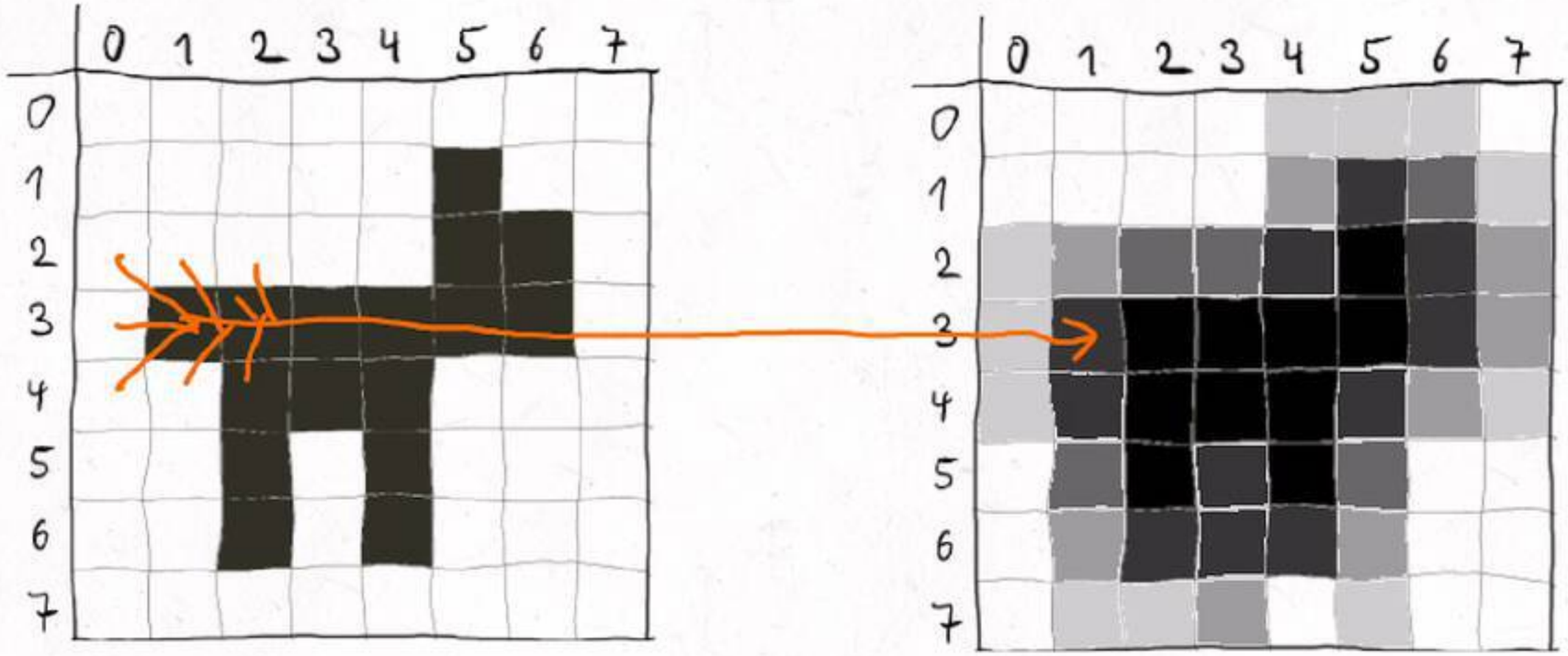
$\frac{1}{9} \cdot$

1	1	1
1	1	1
1	1	1

Box Blur (Unschärfe)

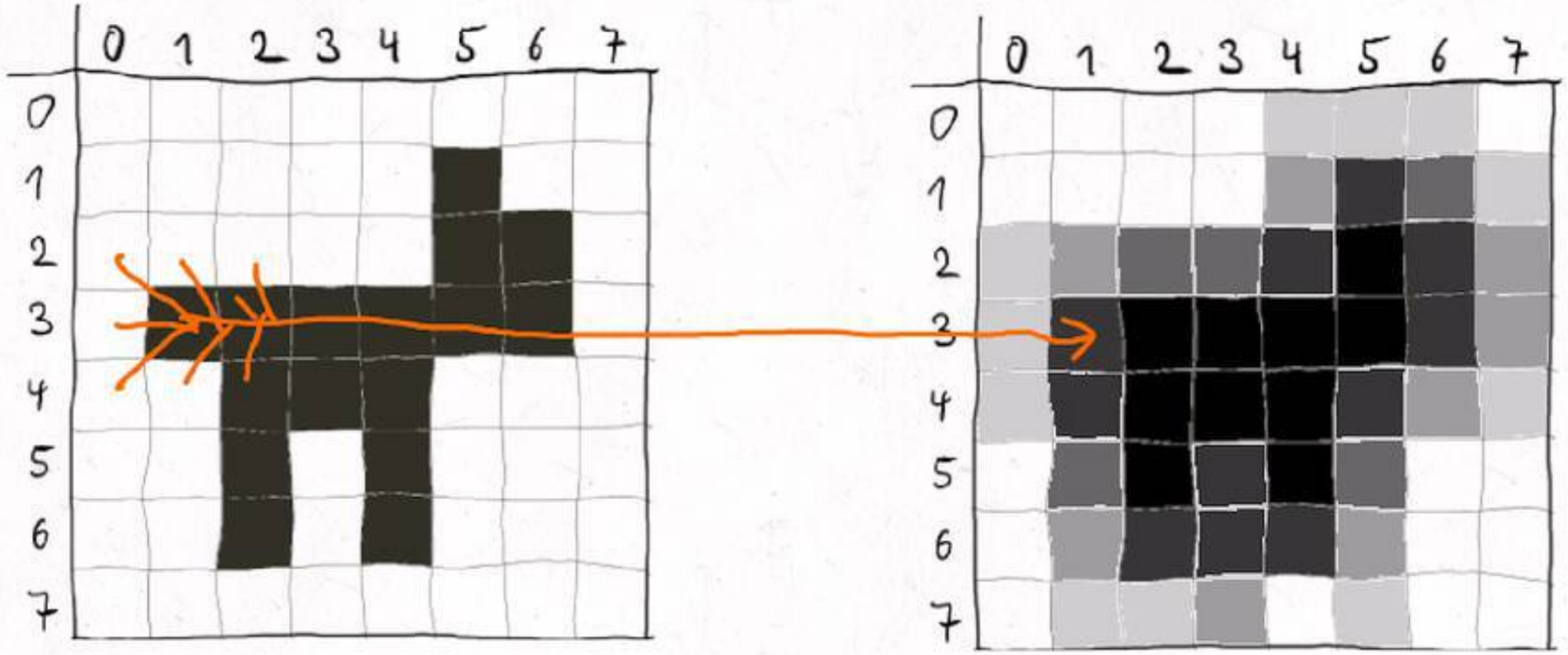


Jeder Zielwert wird aus der Ausgangsmatrix berechnet



Jeder Zielwert wird aus der Ausgangsmatrix berechnet

Jeder Zielwert kann unabhängig berechnet werden



Jeder Zielwert wird aus der Ausgangsmatrix berechnet

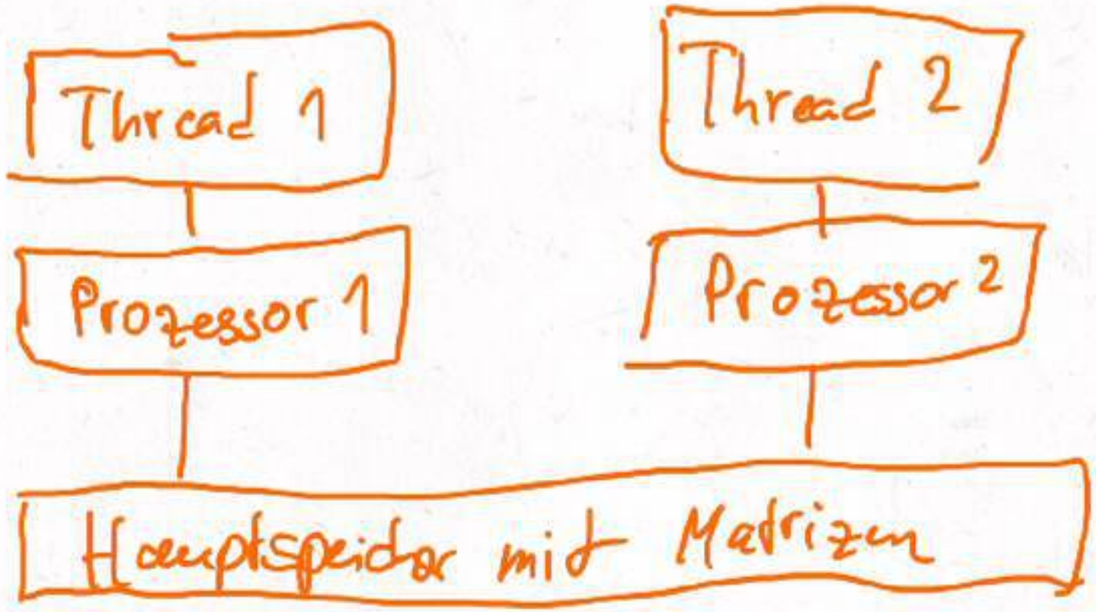
Jeder Zielwert kann unabhängig berechnet werden

Lächerlich einfach zu parallelisieren

Wir weisen die Pixel einfach irgendwie mehreren Threads zu

Wir weisen die Pixel einfach irgendwie mehreren Threads zu

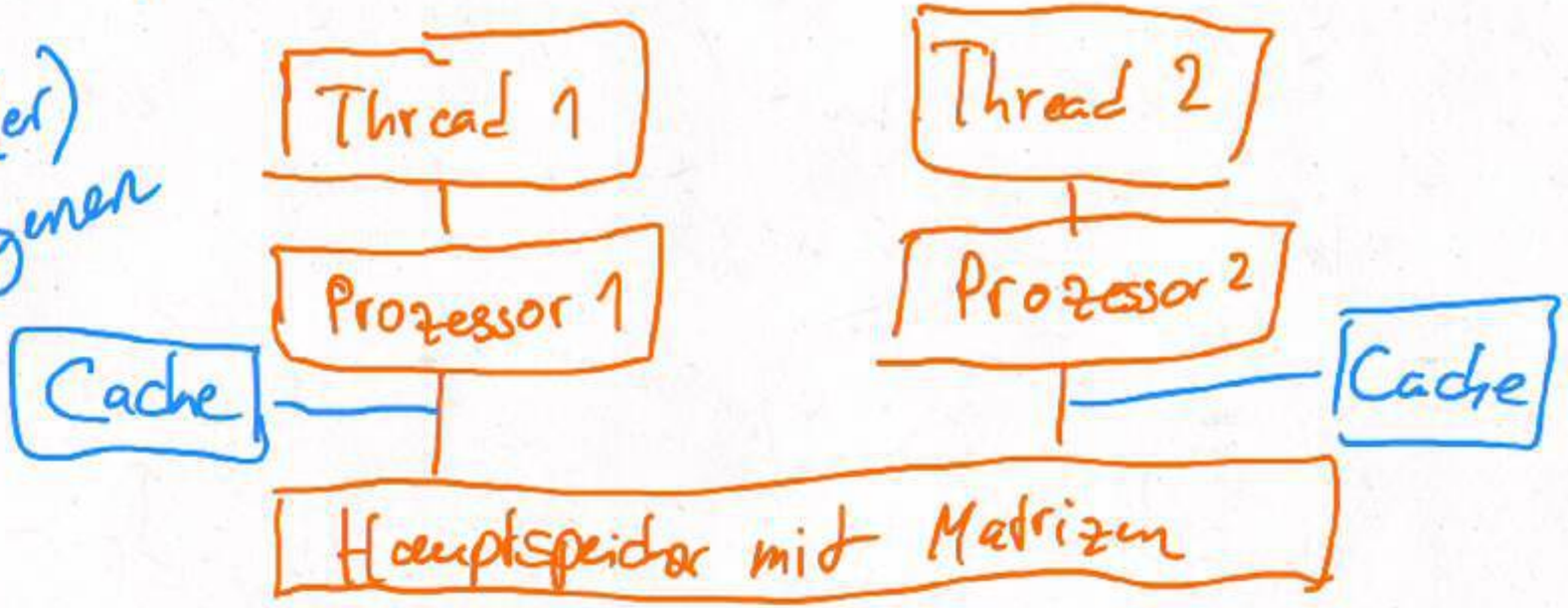
Die Threads laufen dann parallel auf Prozessoren :)



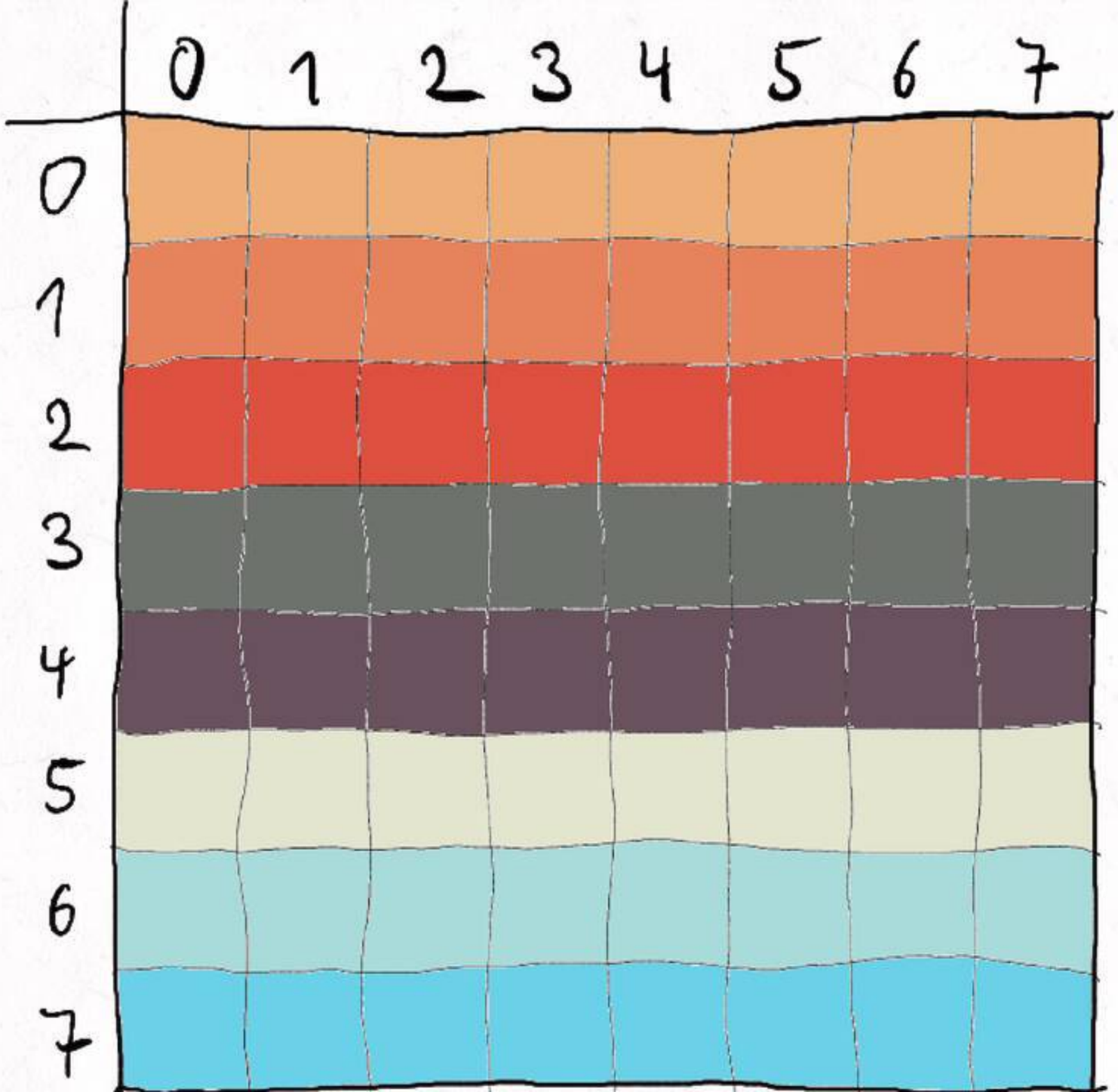
Wir weisen die Pixel einfach irgendwie mehreren Threads zu

Die Threads laufen dann parallel auf Prozessoren :)

Aber:
- Hauptspeicher ist langsam
- Caches sind schnell(er)
- Jeder Prozessor hat eigenen Cache

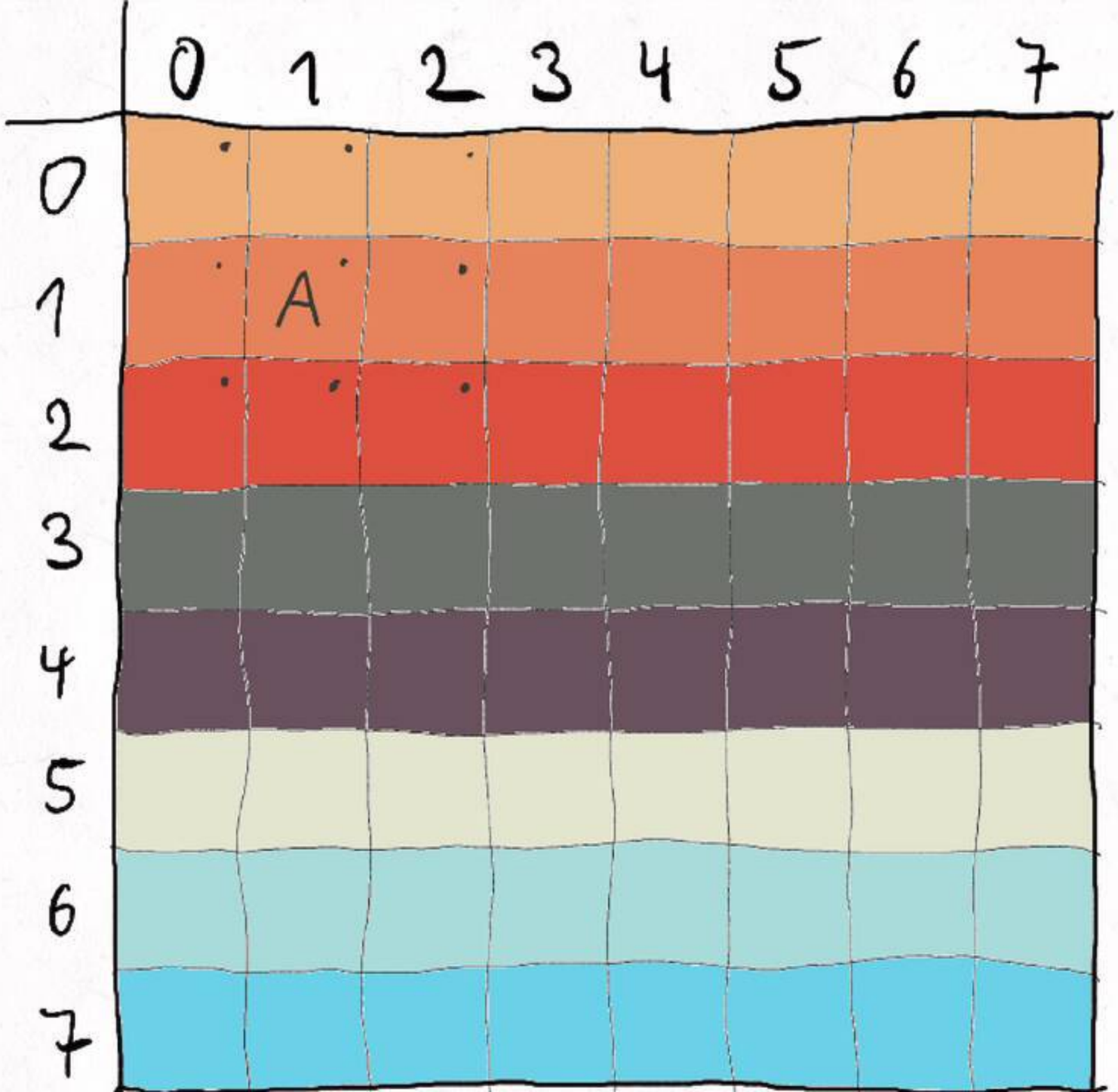


Wie teilen wir die Matrix auf Threads auf, wenn wir die Cache-Nutzung optimieren wollen damit der Hauptspeicher nicht die CPU ausbremst?



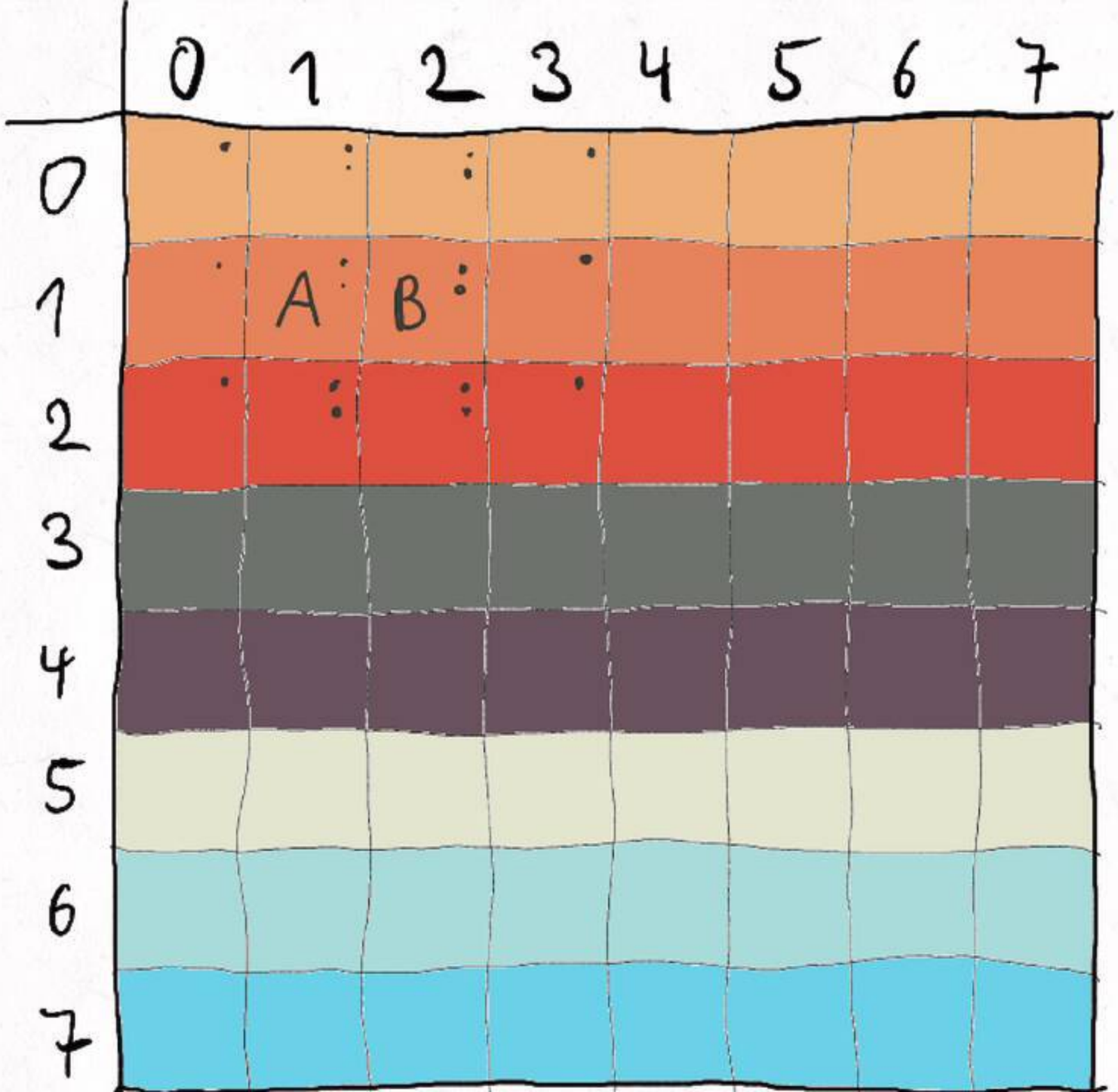
Erster Ansatz:
Zeilenweise
Aufteilung
Jede Farbe stellt
einen anderen
Thread dar

(Hier: 8 Zeilen mit
je 8 Spalten und
8 Threads / CPUs
... wie praktisch)



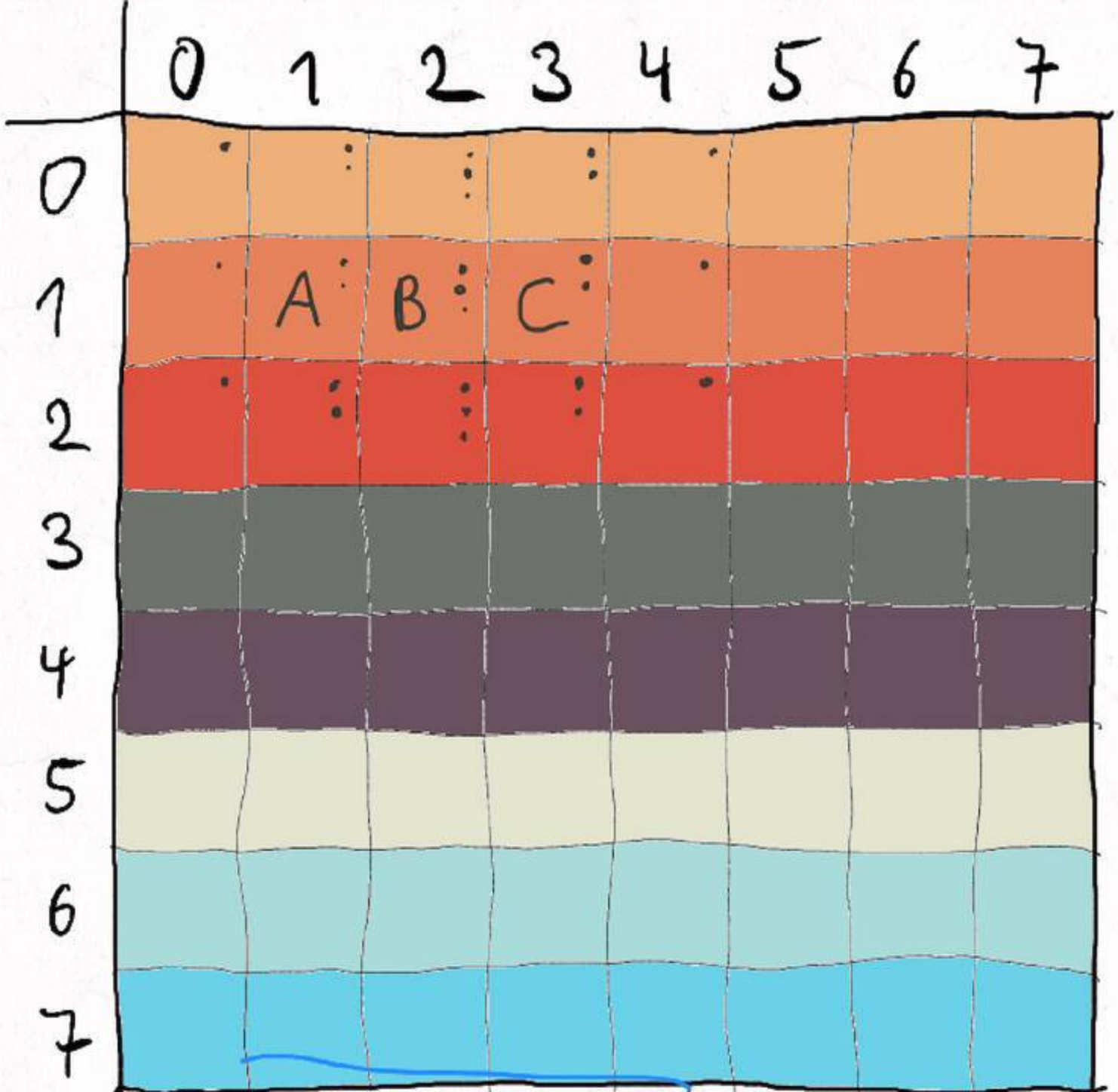
Thread 1

New	Cached
A	9
	Ø



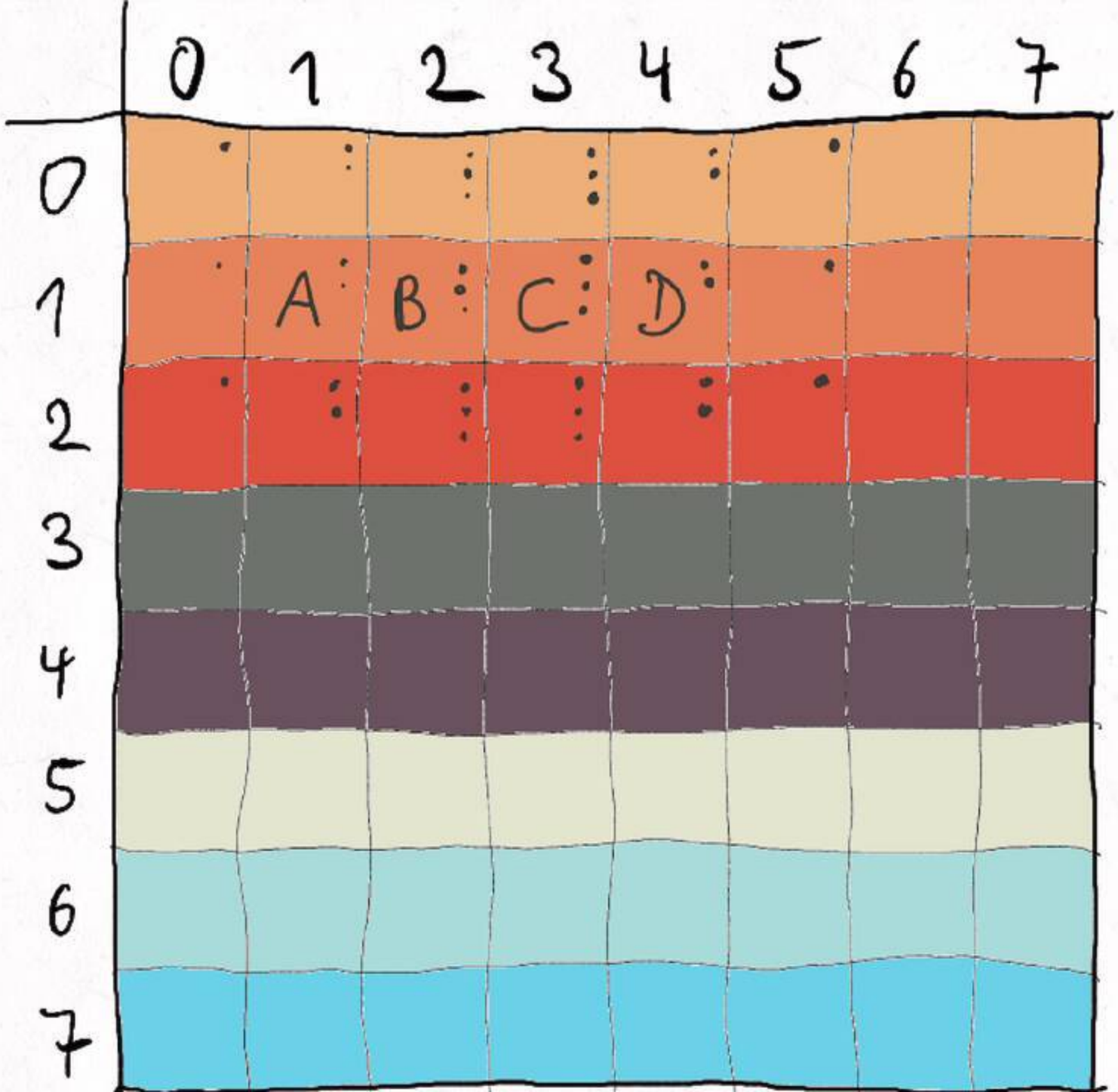
Thread 1

	New	Cached
A	9	∅
B	3	6



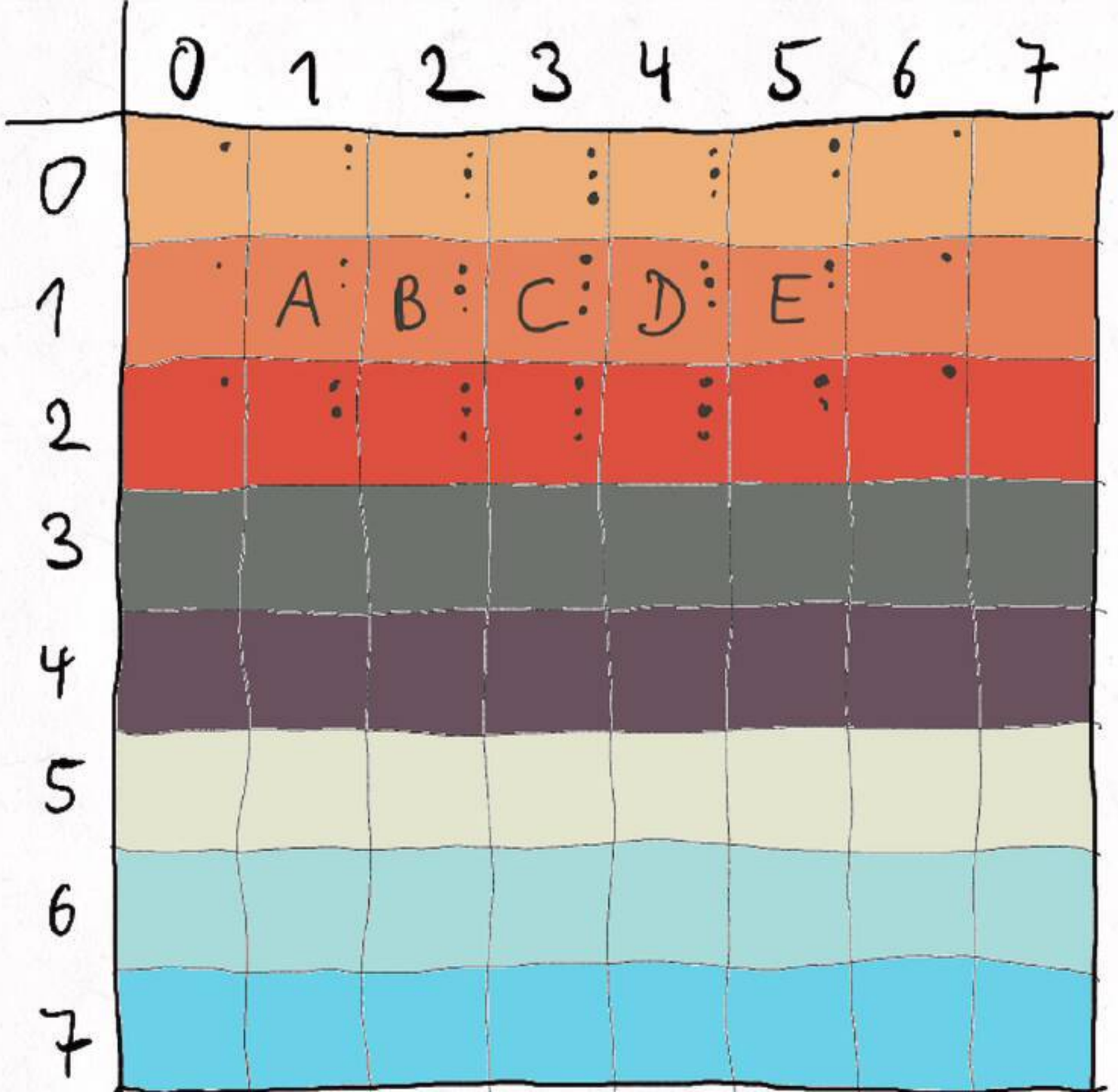
Thread 1

	New	Cached
A	9	∅
B	3	6
C	3	6



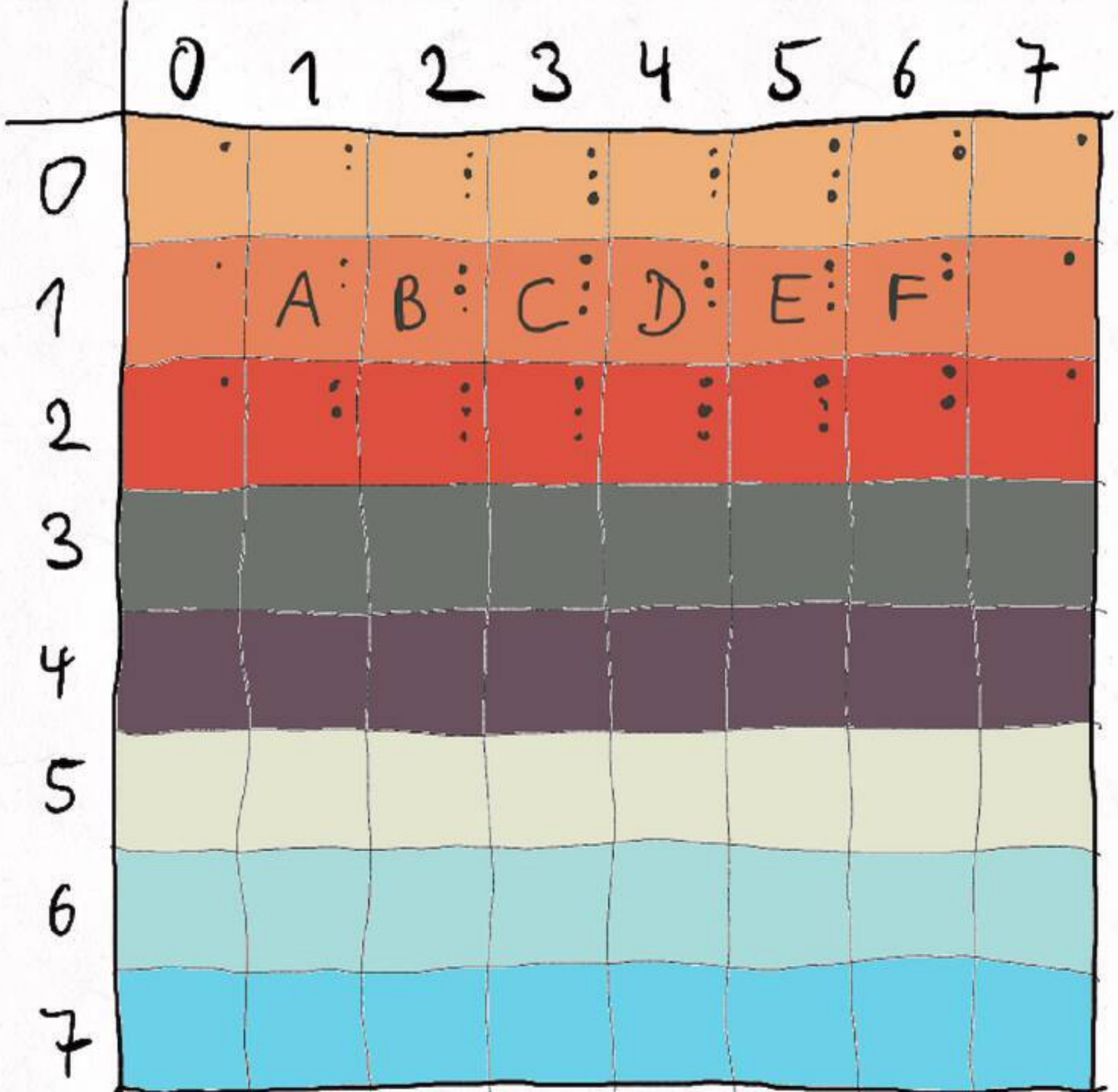
Thread 1

	New	Cached
A	9	∅
B	3	6
C	3	6
D	3	6



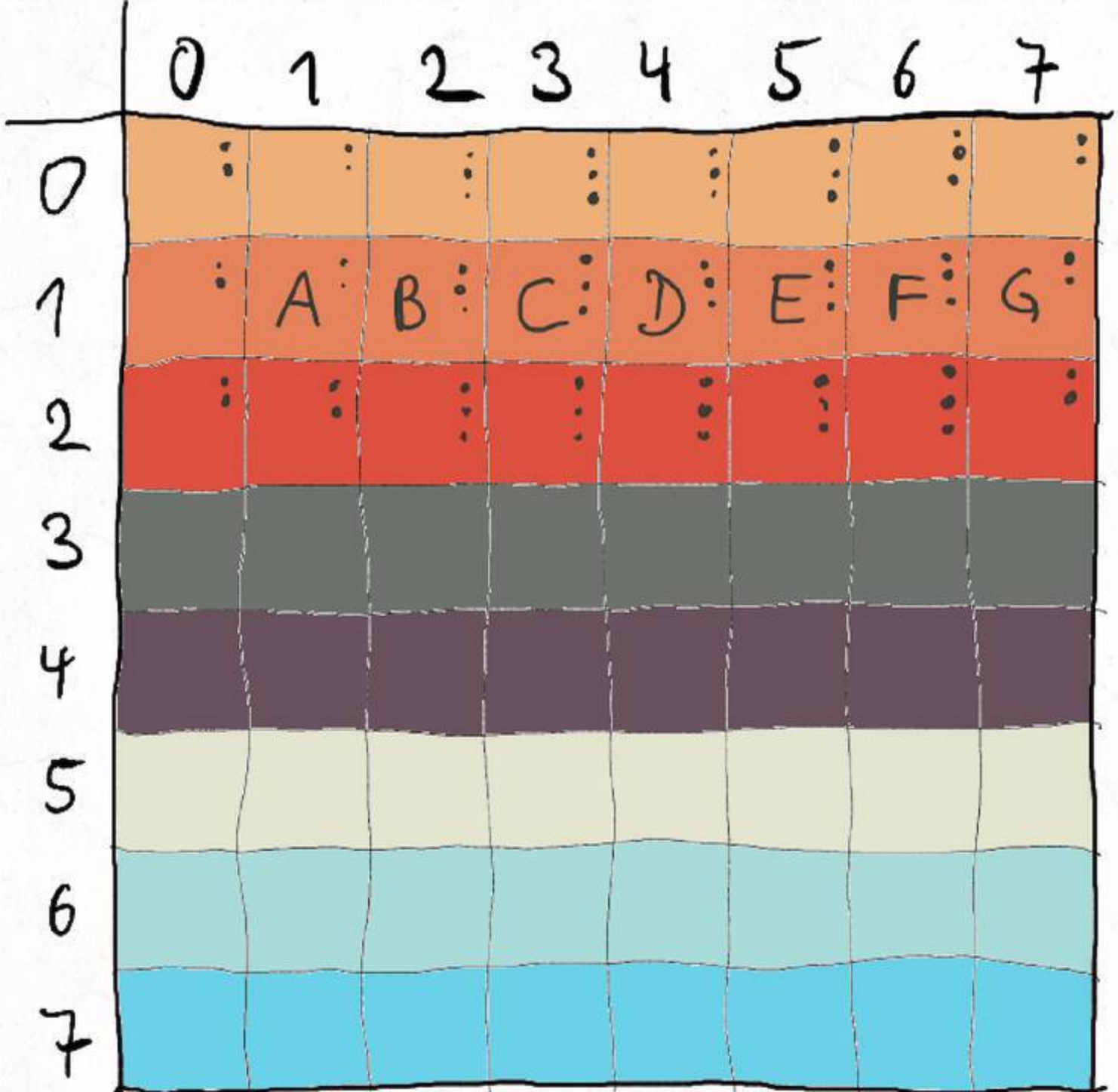
Thread 1

	New	Cached
A	9	∅
B	3	6
C	3	6
D	3	6
E	3	6



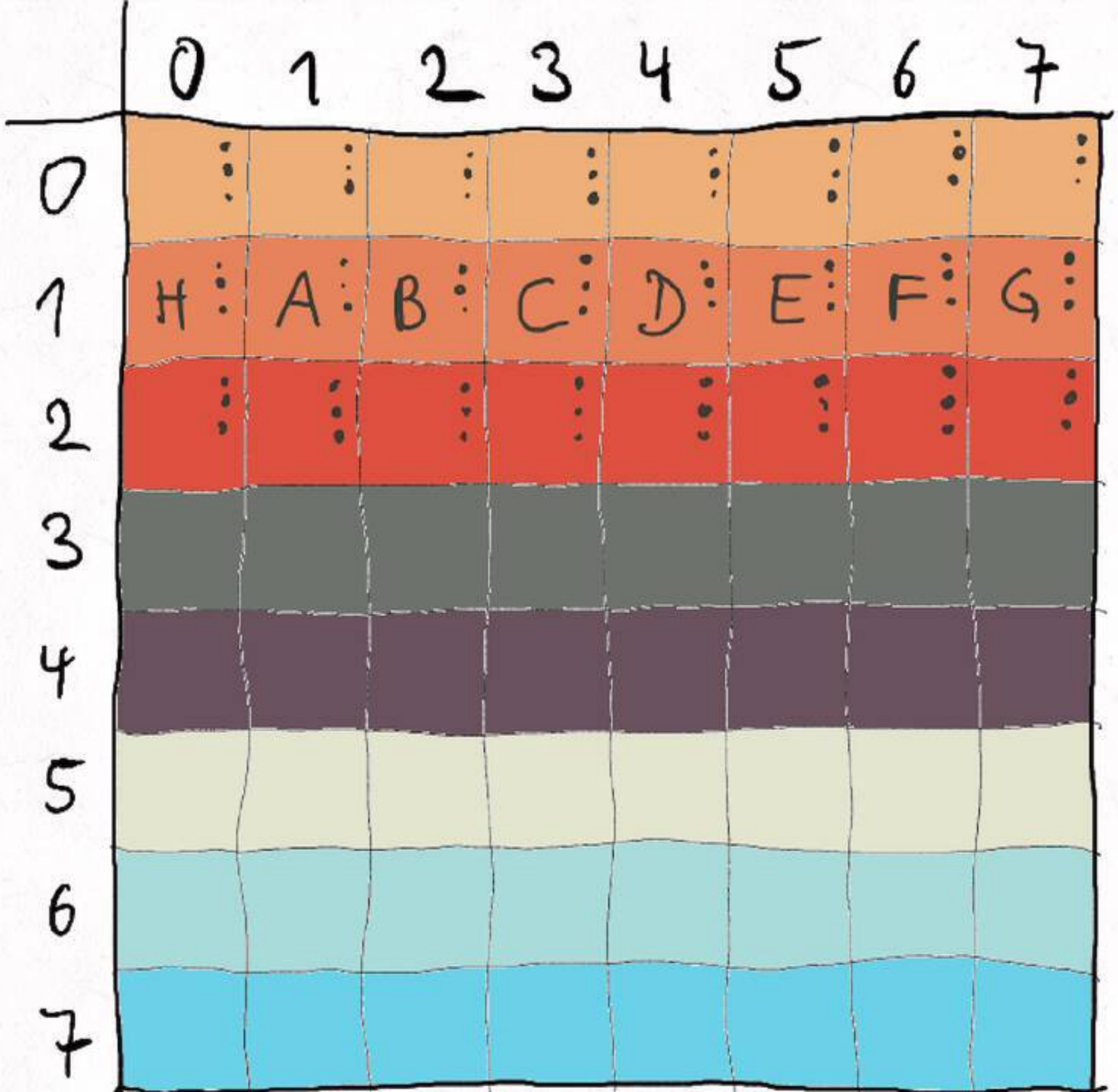
Thread 1

	New	Cached
A	9	∅
B	3	6
C	3	6
D	3	6
E	3	6
F	3	6



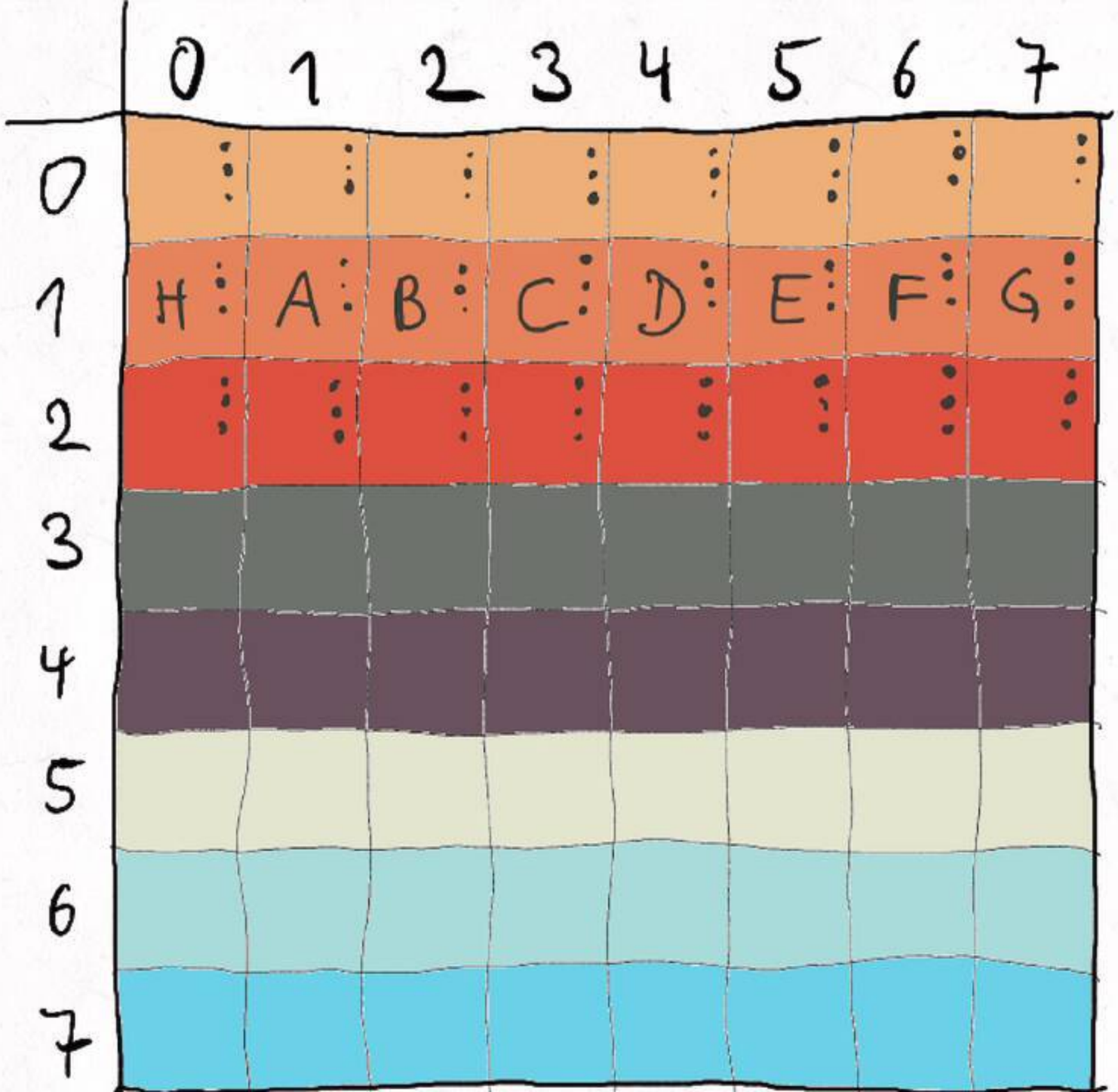
Thread 1

	New	Cached
A	9	0
B	3	6
C	3	6
D	3	6
E	3	6
F	3	6
G	0	9



Thread 1

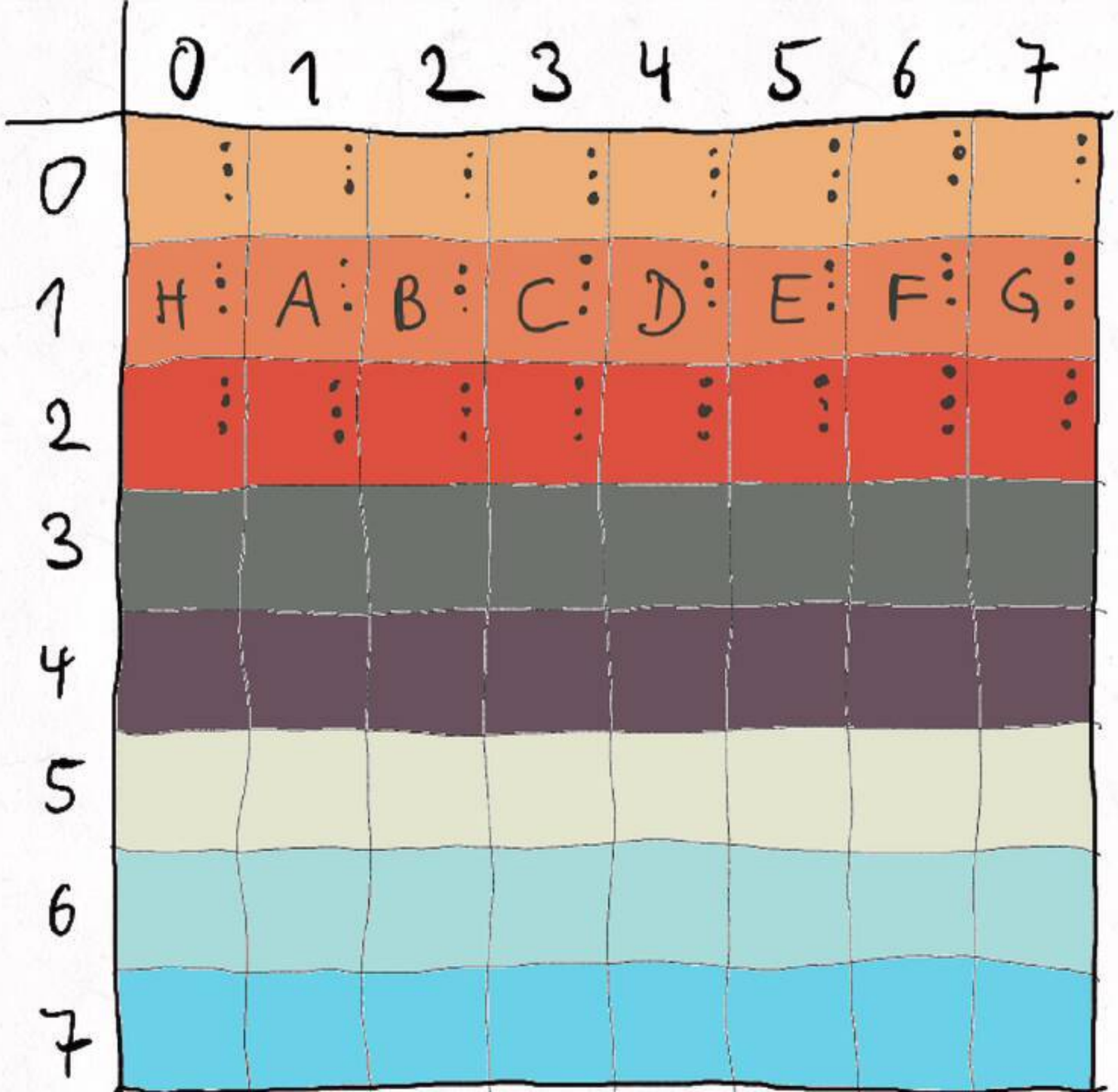
	New	Cached
A	9	Ø
B	3	6
C	3	6
D	3	6
E	3	6
F	3	6
G	0	9
H	0	9



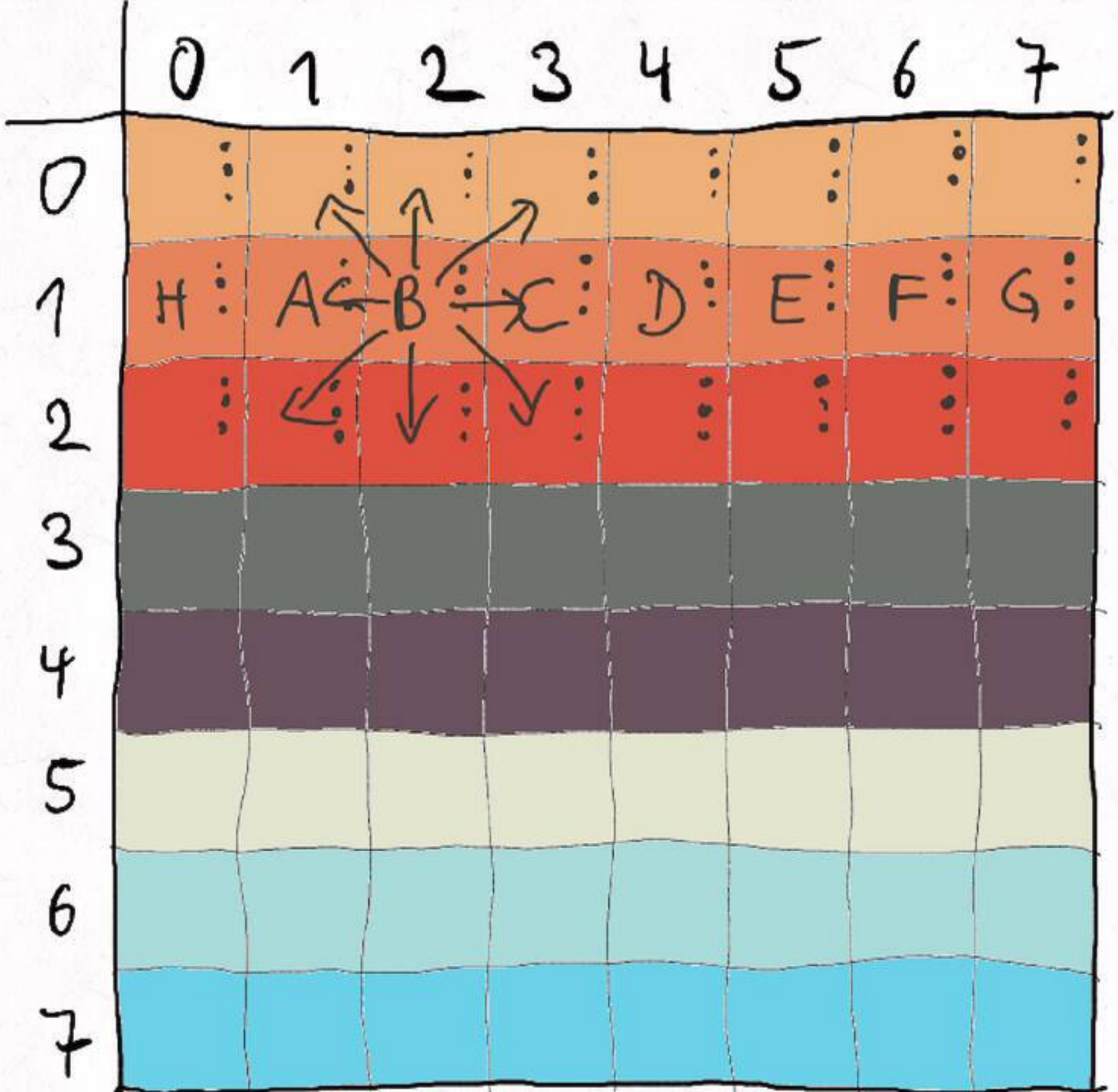
Thread 1

	New	Cached	
A	9	0	
B	3	6	
C	3	6	
D	3	6	
E	3	6	
F	3	6	
G	0	9	
H	0	9	8×9

$24 + 48 = 72$



Thread 1
24 Werte aus Speicher
in Cache geladen

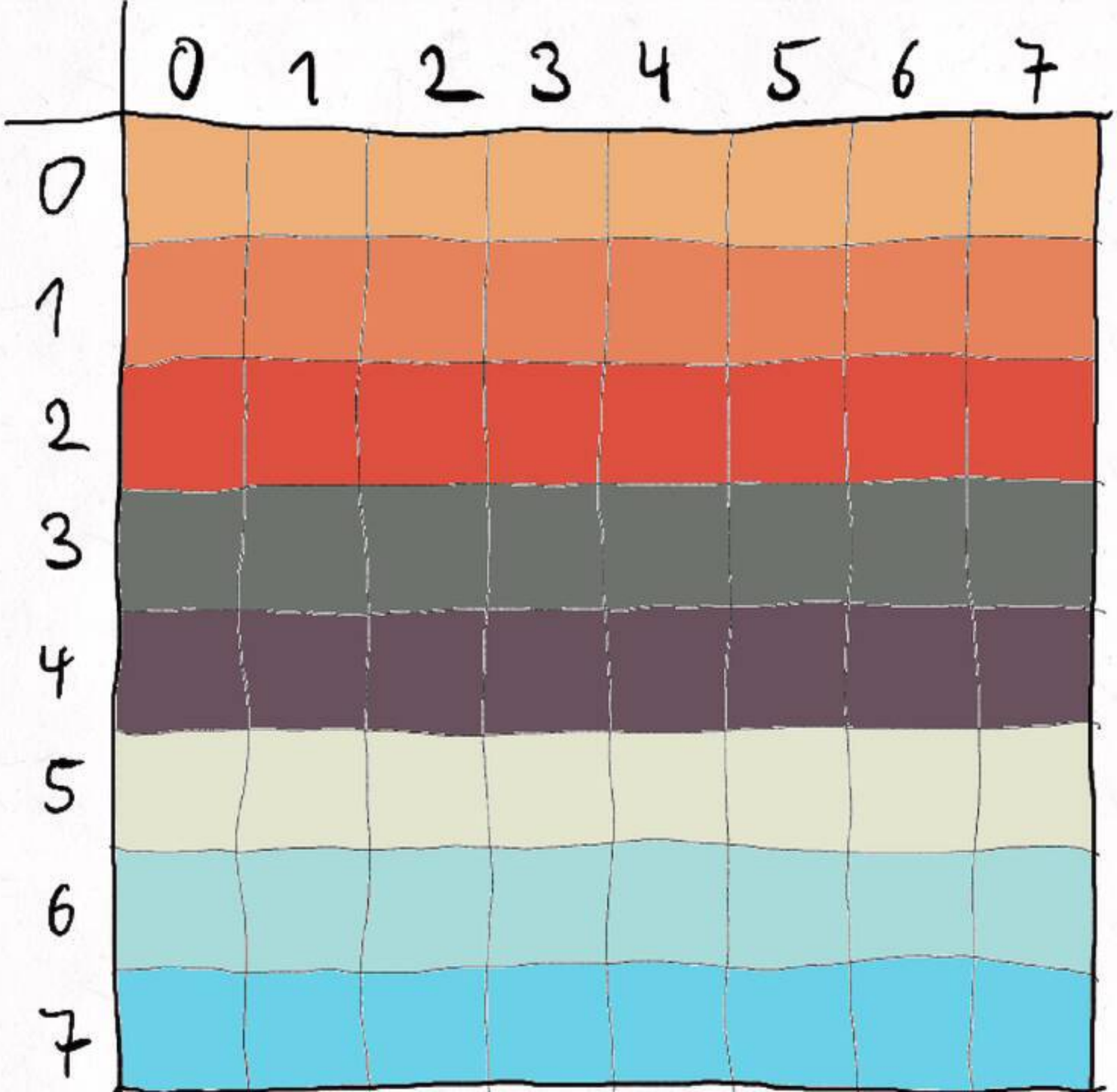


Thread 1
24 Werte aus Speicher in Cache geladen

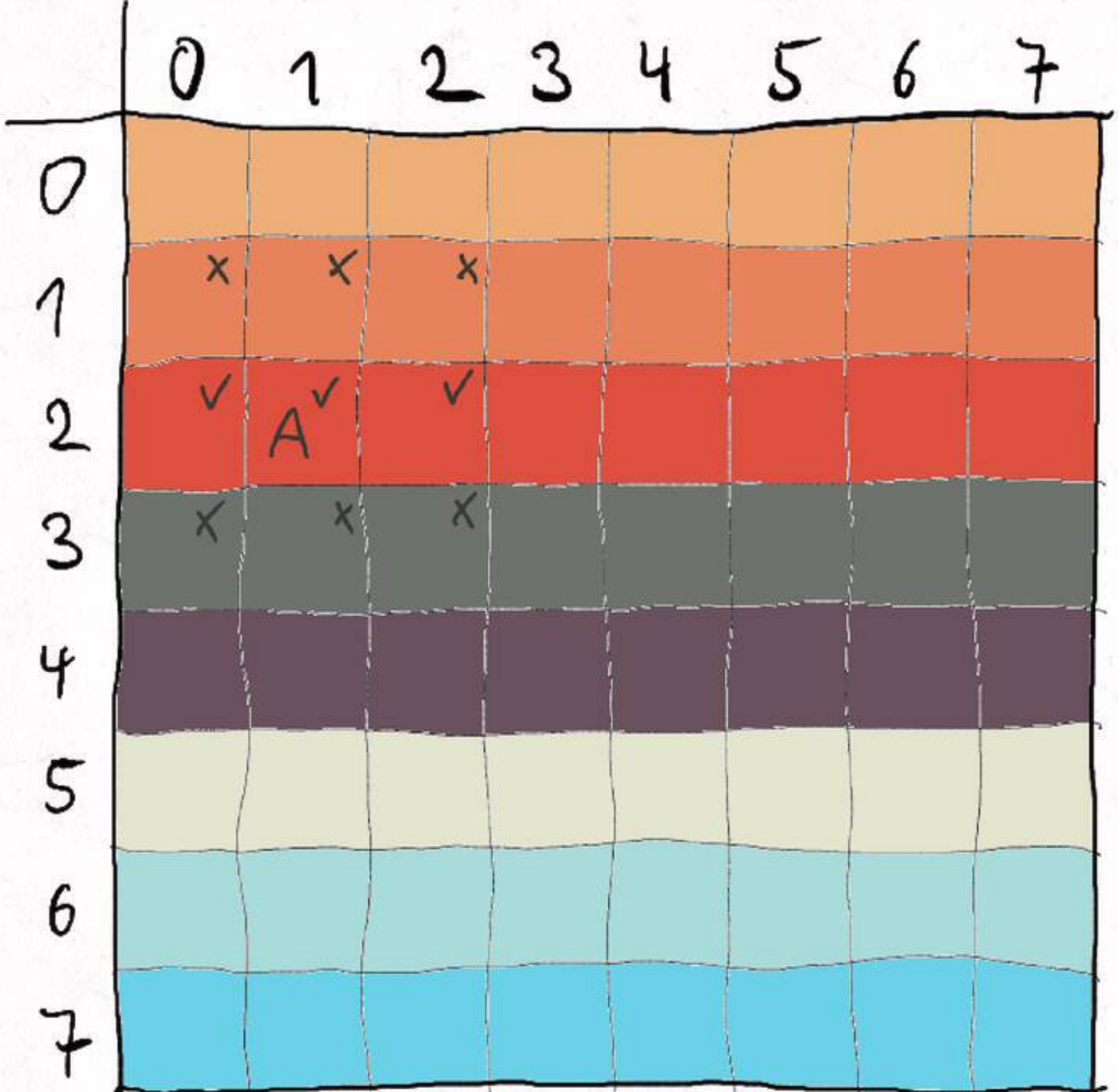
Jeder Wert wird auch noch in 2 weiteren Threads genutzt

Jeder Thread (1 pro Kern) hat eigenen Cache
Muss also erneut gelesen werden

8 Threads x 24 Reads = 192

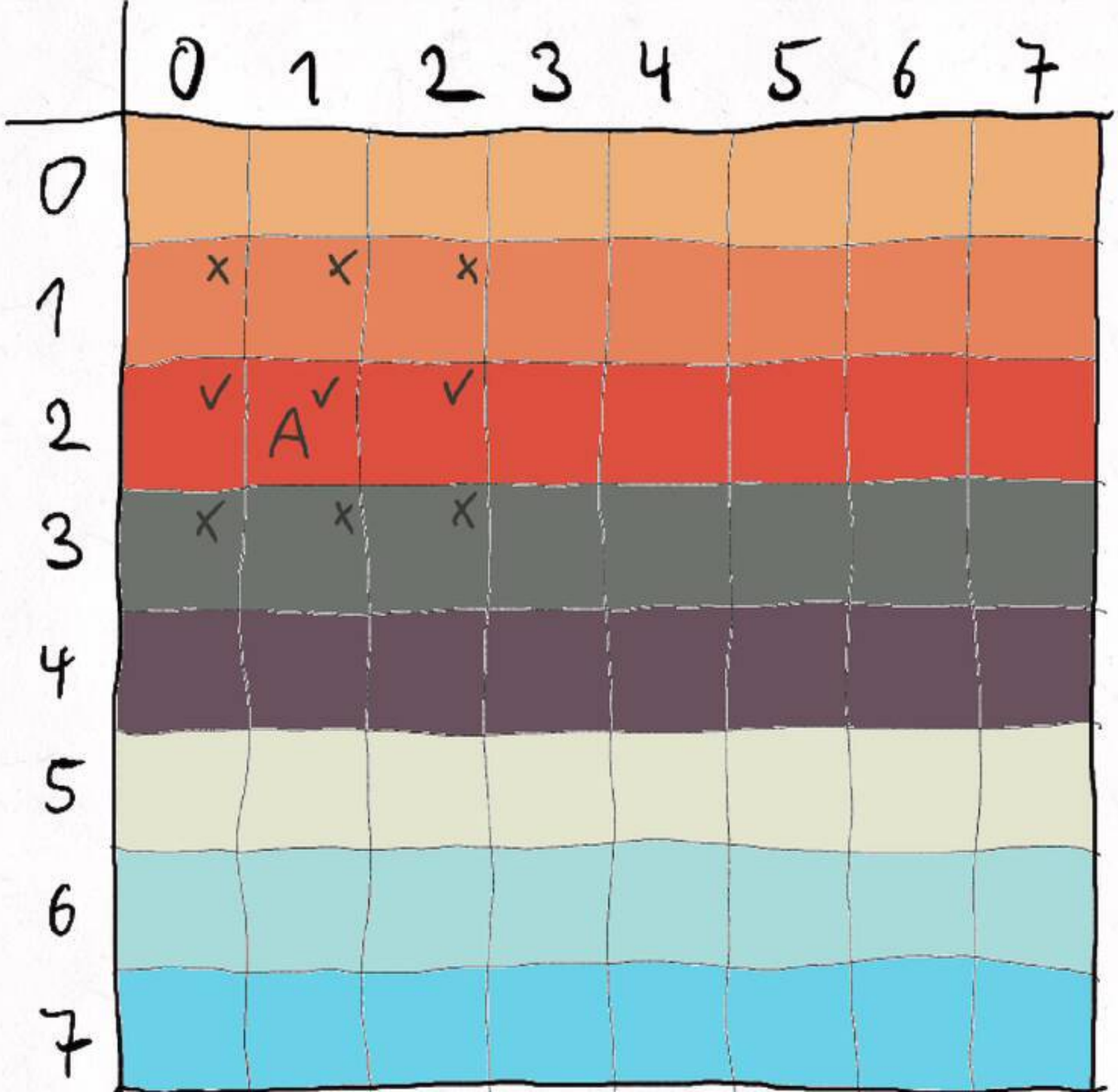


- Geschrieben wird in separates Ziel-Array
- Austausch Ziel- und Quell-Array
- Erneute Iteration



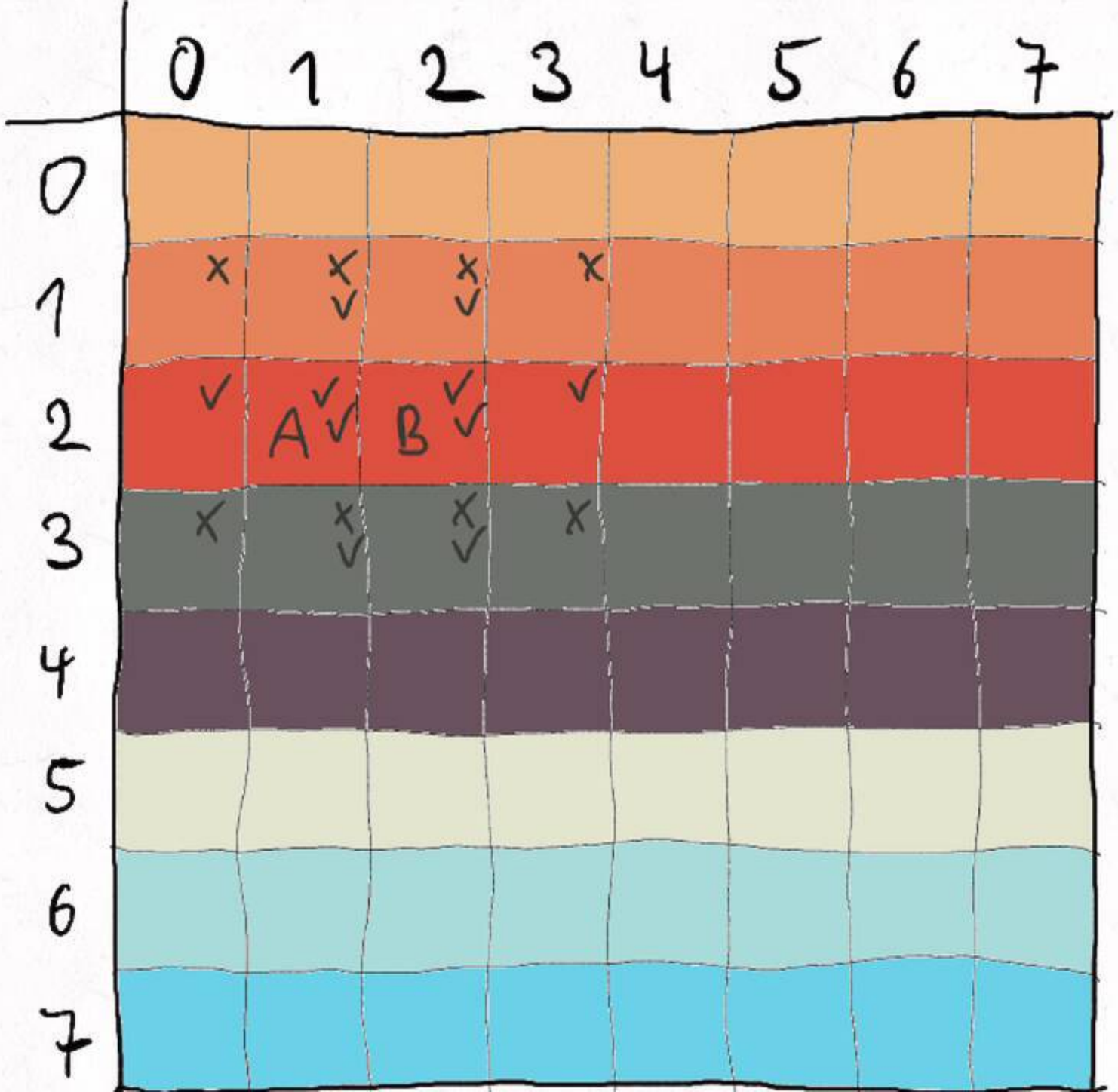
- Geschrieben wird in separates Ziel-Array
- Austausch Ziel- und Quell-Array
- Erneute Iteration

Im eigenen Thread berechnete Werte sind im Cache – Rest muss gelesen werden



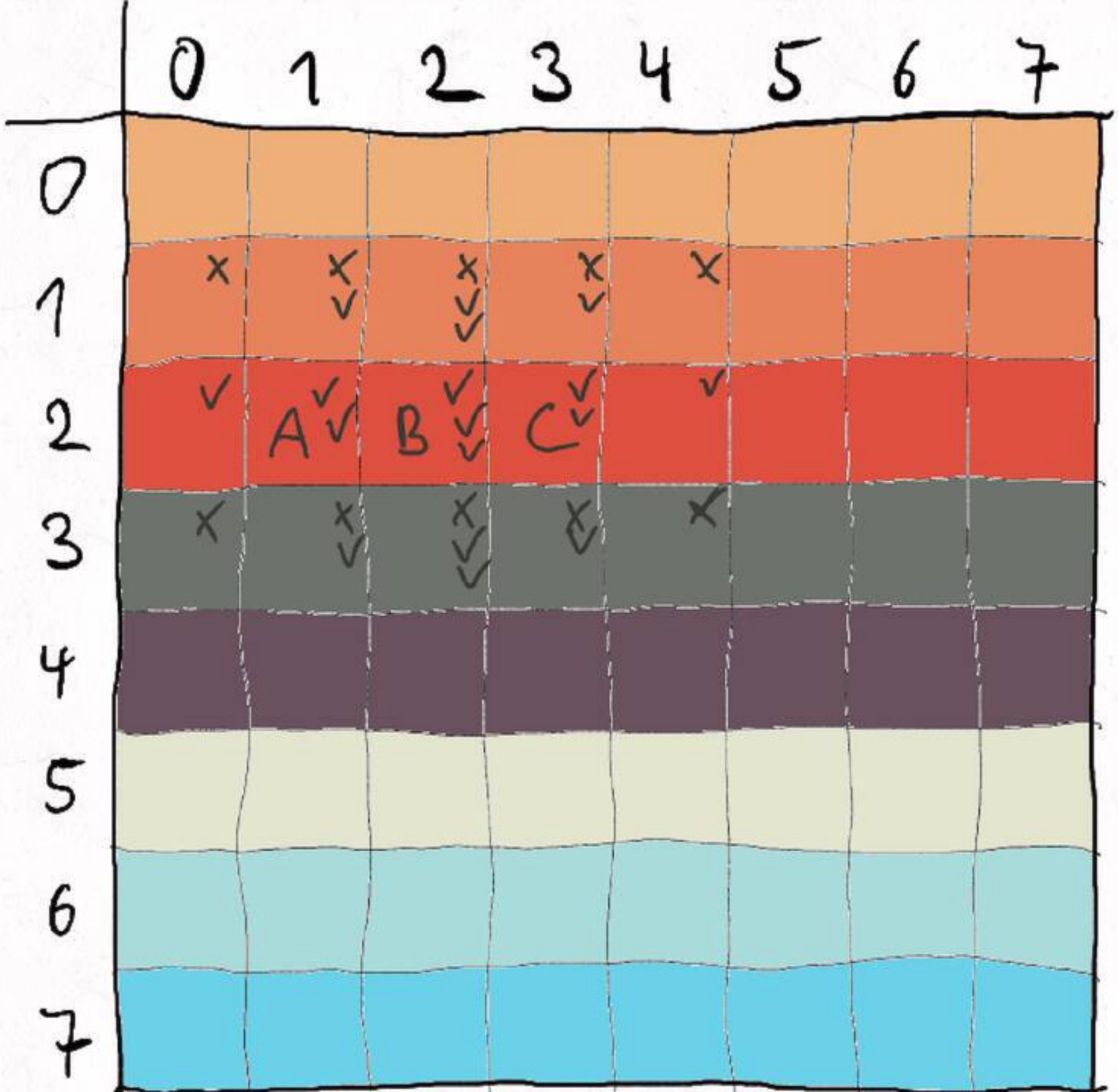
Im eigenen Thread berechnete Werte sind im Cache – Rest muss gelesen werden

	New	Cache
A	6	3



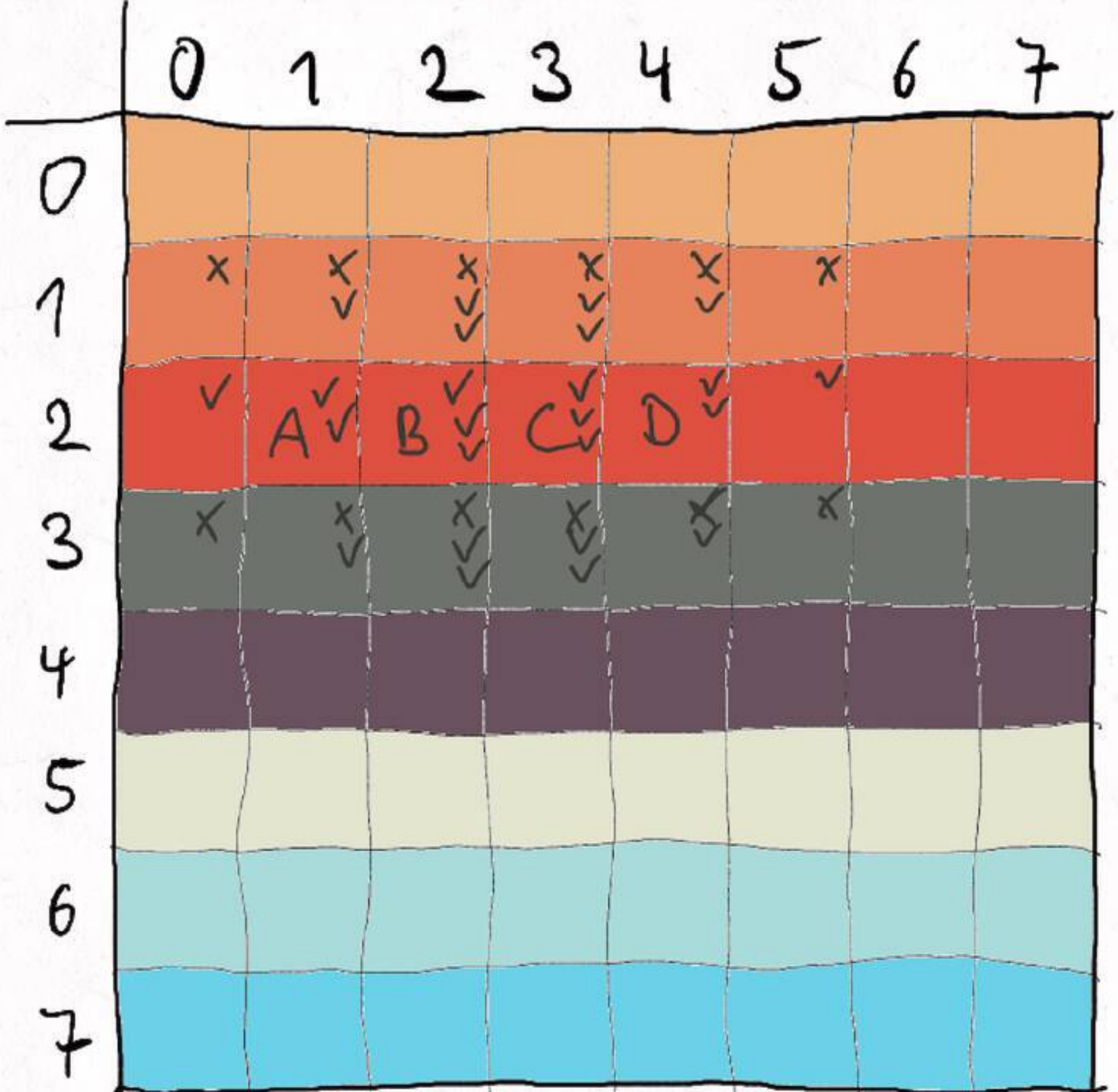
Im eigenen Thread berechnete Werte sind im Cache – Rest muss gelesen werden

	New	Cache
A	6	3
B	2	7



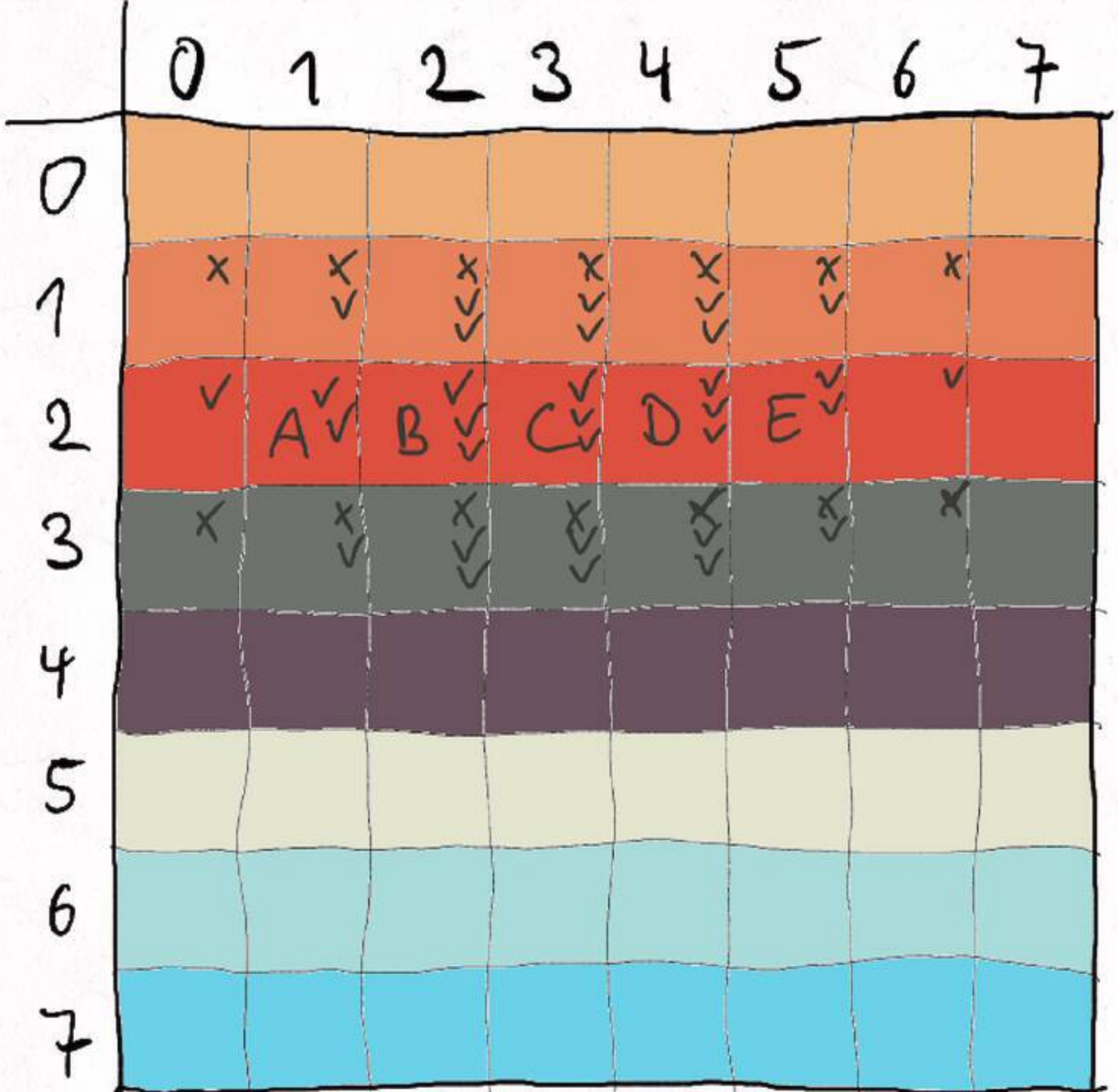
Im eigenen Thread berechnete Werte sind im Cache – Rest muss gelesen werden

	New	Cache
A	6	3
B	2	7
C	2	7



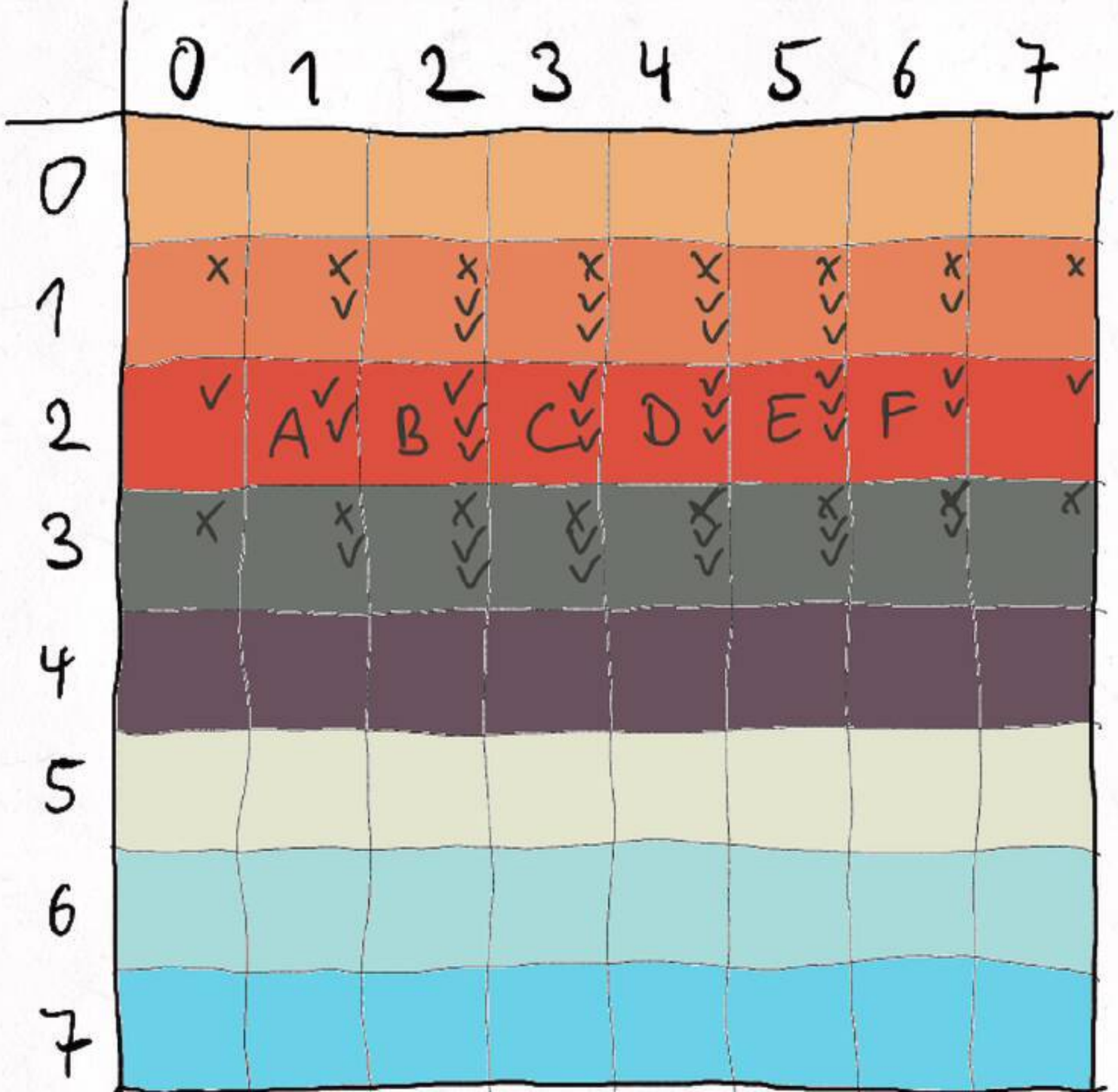
Im eigenen Thread berechnete Werte sind im Cache – Rest muss gelesen werden

	New	Cache
A	6	3
B	2	7
C	2	7
D	2	7



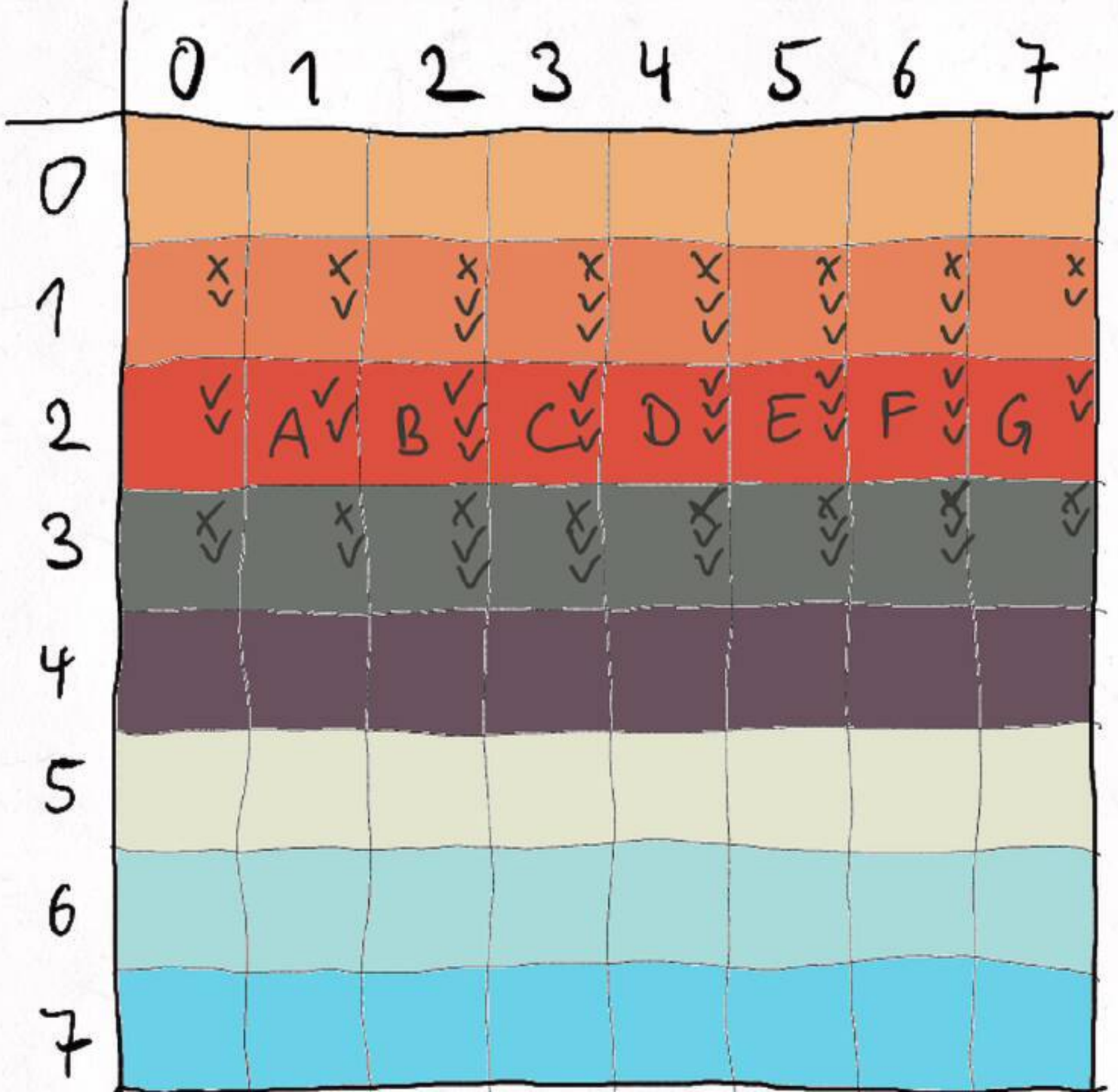
Im eigenen Thread berechnete Werte sind im Cache – Rest muss gelesen werden

	New	Cache
A	6	3
B	2	7
C	2	7
D	2	7
E	2	2



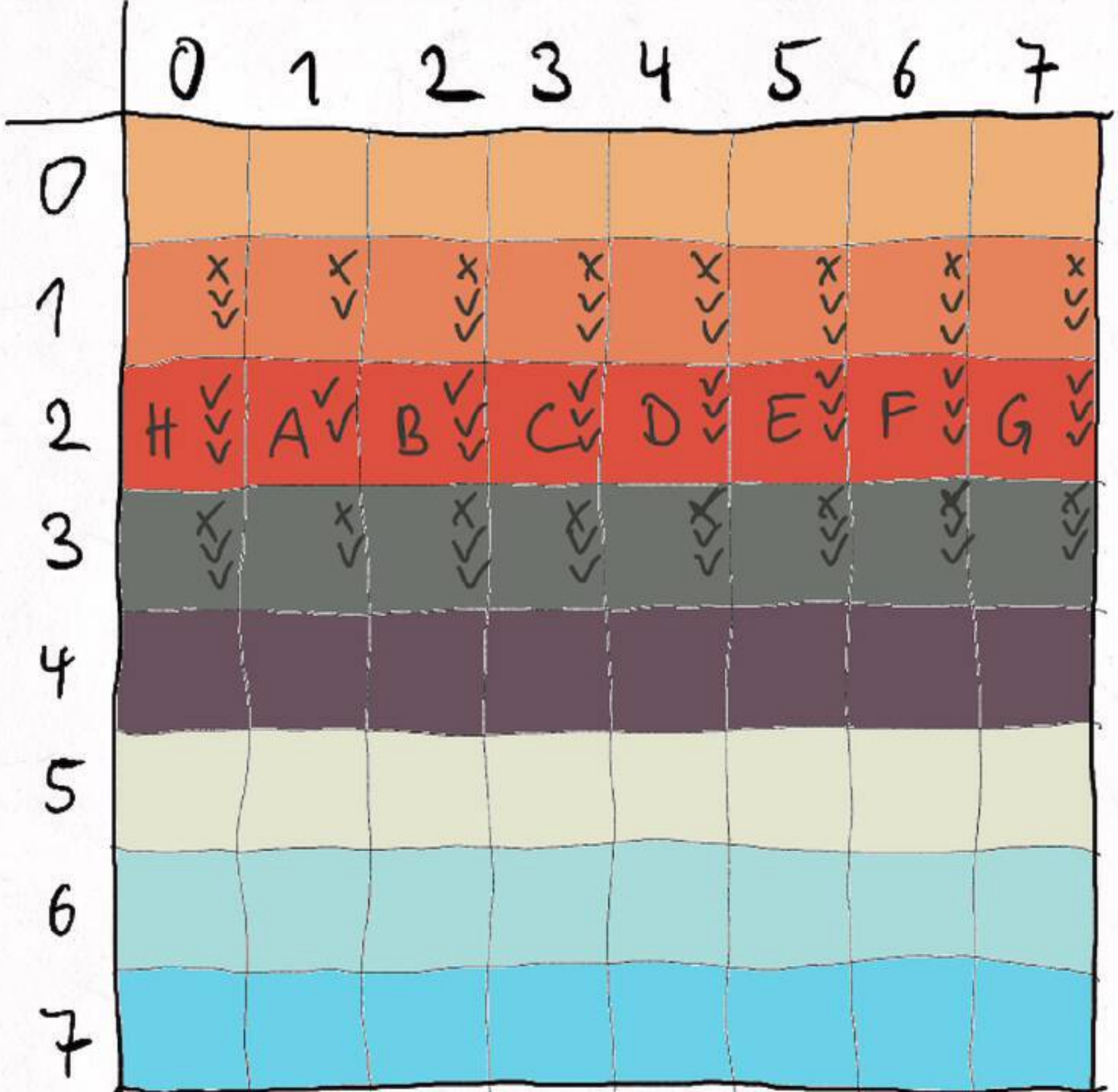
Im eigenen Thread berechnete Werte sind im Cache – Rest muss gelesen werden

	New	Cache
A	6	3
B	2	7
C	2	7
D	2	7
E	2	7
F	2	7



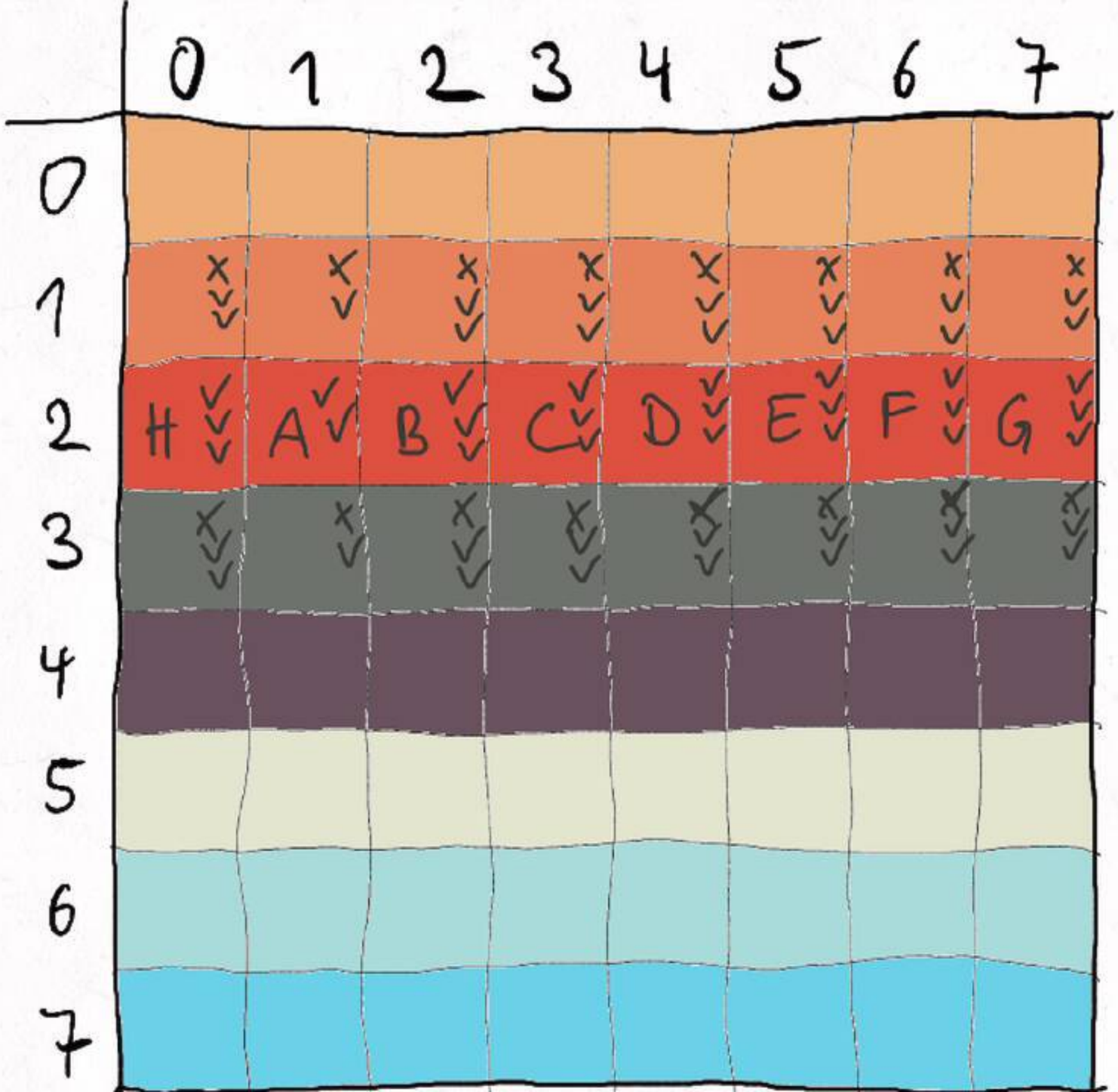
Im eigenen Thread berechnete Werte sind im Cache – Rest muss gelesen werden

	New	Cache
A	6	3
B	2	7
C	2	7
D	2	7
E	2	7
F	2	7
G	0	9



Im eigenen Thread berechnete Werte sind im Cache – Rest muss gelesen werden

	New	Cache
A	6	3
B	2	7
C	2	7
D	2	7
E	2	7
F	2	7
G	0	9
H	0	9



Im eigenen Thread berechnete Werte sind im Cache – Rest muss gelesen werden

	New	Cache
A	6	3
B	2	7
C	2	7
D	2	7
E	2	7
F	2	7
G	0	9
H	0	9

16 + 56 = 72 (8×9)

Wie compute-intensive ist unser Ansatz?

$8 \times 16 = 128$ Reads pro Durchgang

Wie compute-intensive ist unser Ansatz?

$8 \times 16 = 128$ Reads pro Durchgang

$+ \quad 8 \times 8 = 64$ Writes pro Durchgang

jeder Wert wird von einem anderen Thread ausgelesen – muss also in den Speicher

Wie compute-intensive ist unser Ansatz?

$8 \times 16 = 128$ Reads pro Durchgang

$+ \quad 8 \times 8 = 64$ Writes pro Durchgang

jeder Wert wird von einem anderen Thread ausgelesen – muss also in den Speicher

$192 \times$ 4 Bytes (Single Prec. Float) = 768 Bytes

Floating Point Ops: 64 Felder $\times$ (1 Multiplikation + 8 Additionen) = 576

Wie compute-intensive ist unser Ansatz?

$8 \times 16 = 128$ Reads pro Durchgang

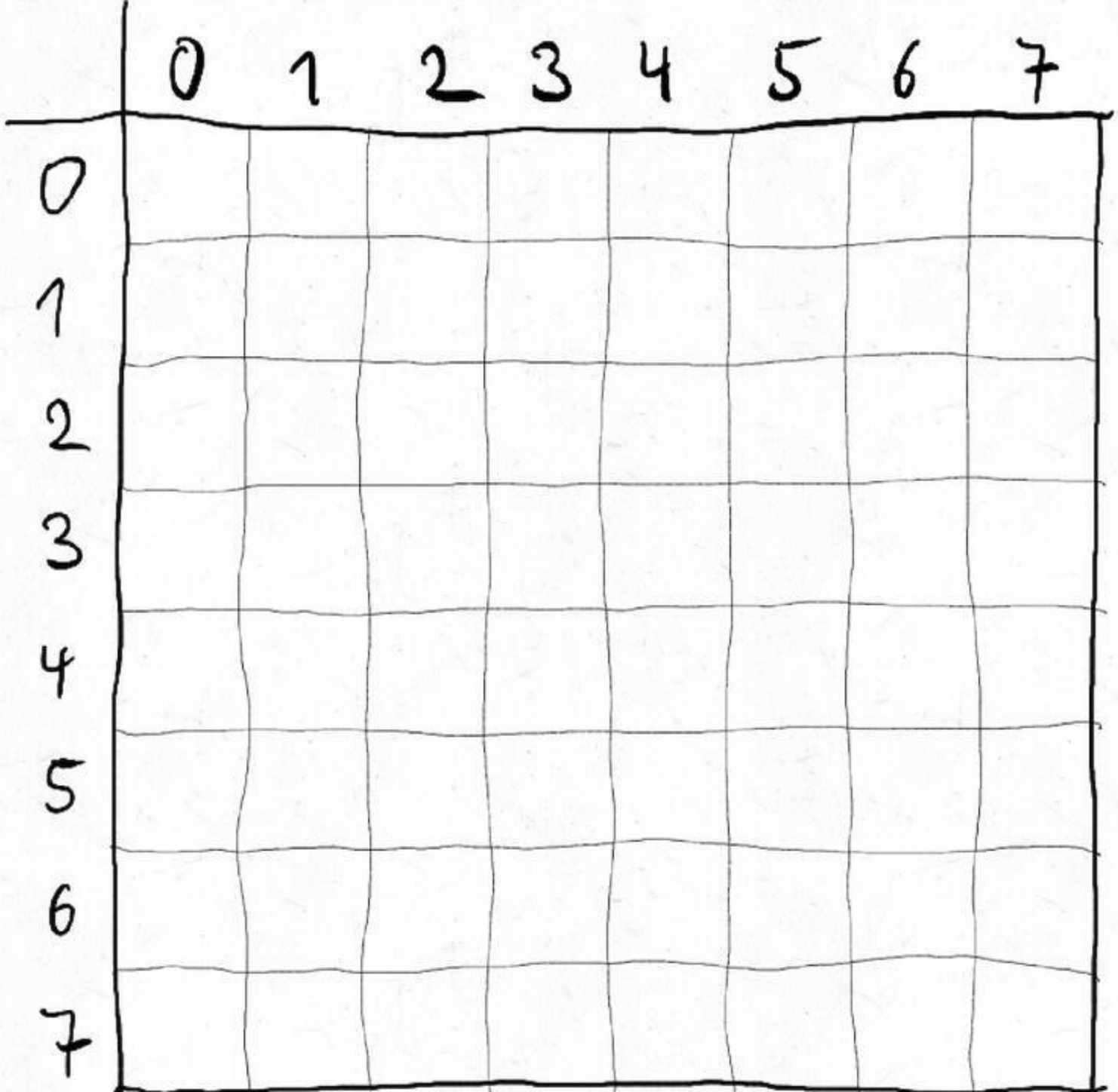
$+ \quad 8 \times 8 = 64$ Writes pro Durchgang

jeder Wert wird von einem anderen Thread ausgelesen – muss also in den Speicher

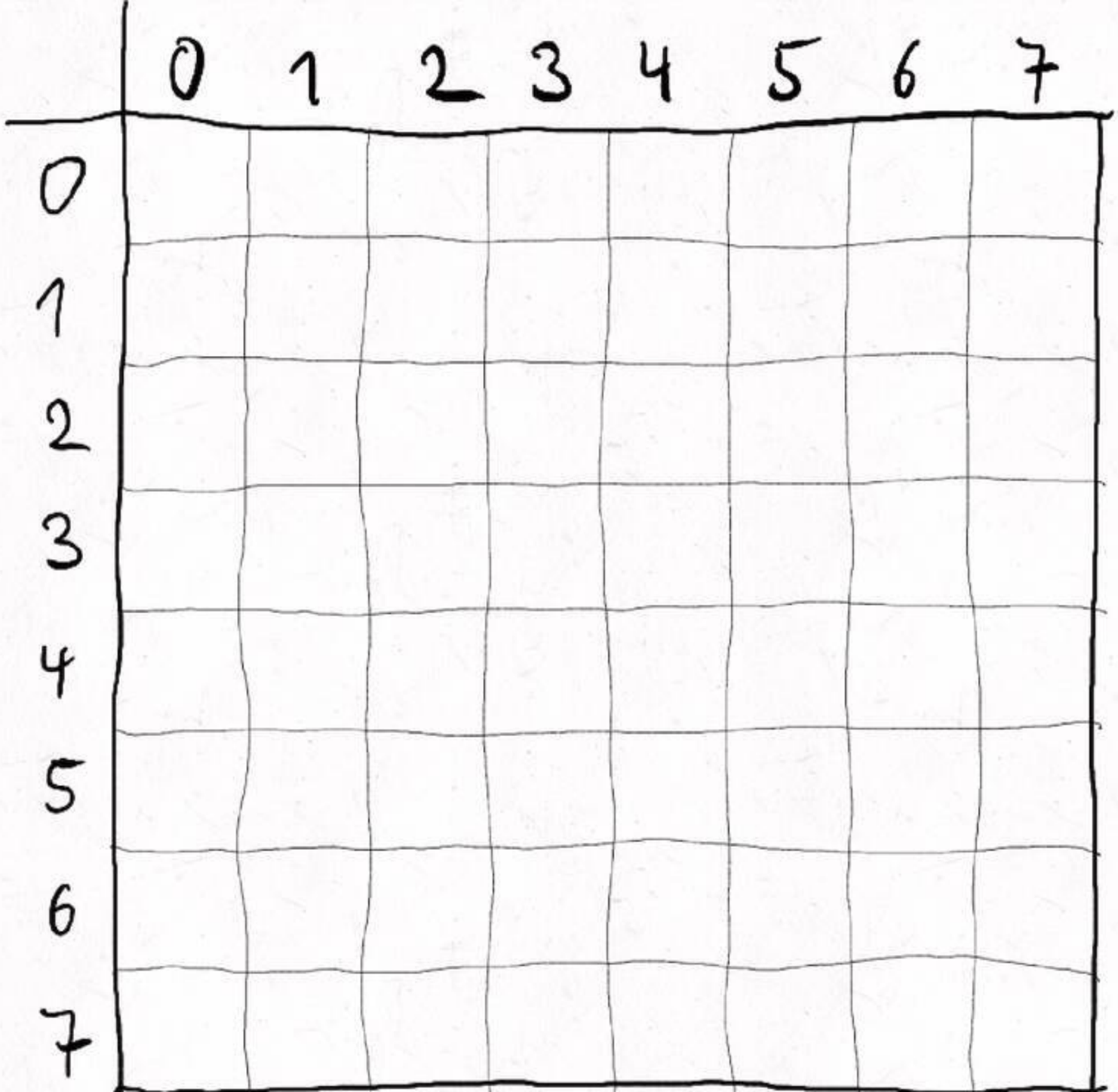
192×4 Bytes (Single Prec. Float) $= 768$ Bytes

Floating Point Ops: 64 Felder $\times$ (1 Multiplikation + 8 Additionen) $= 576$

Intensität: $576 / 768 = 0{,}75$

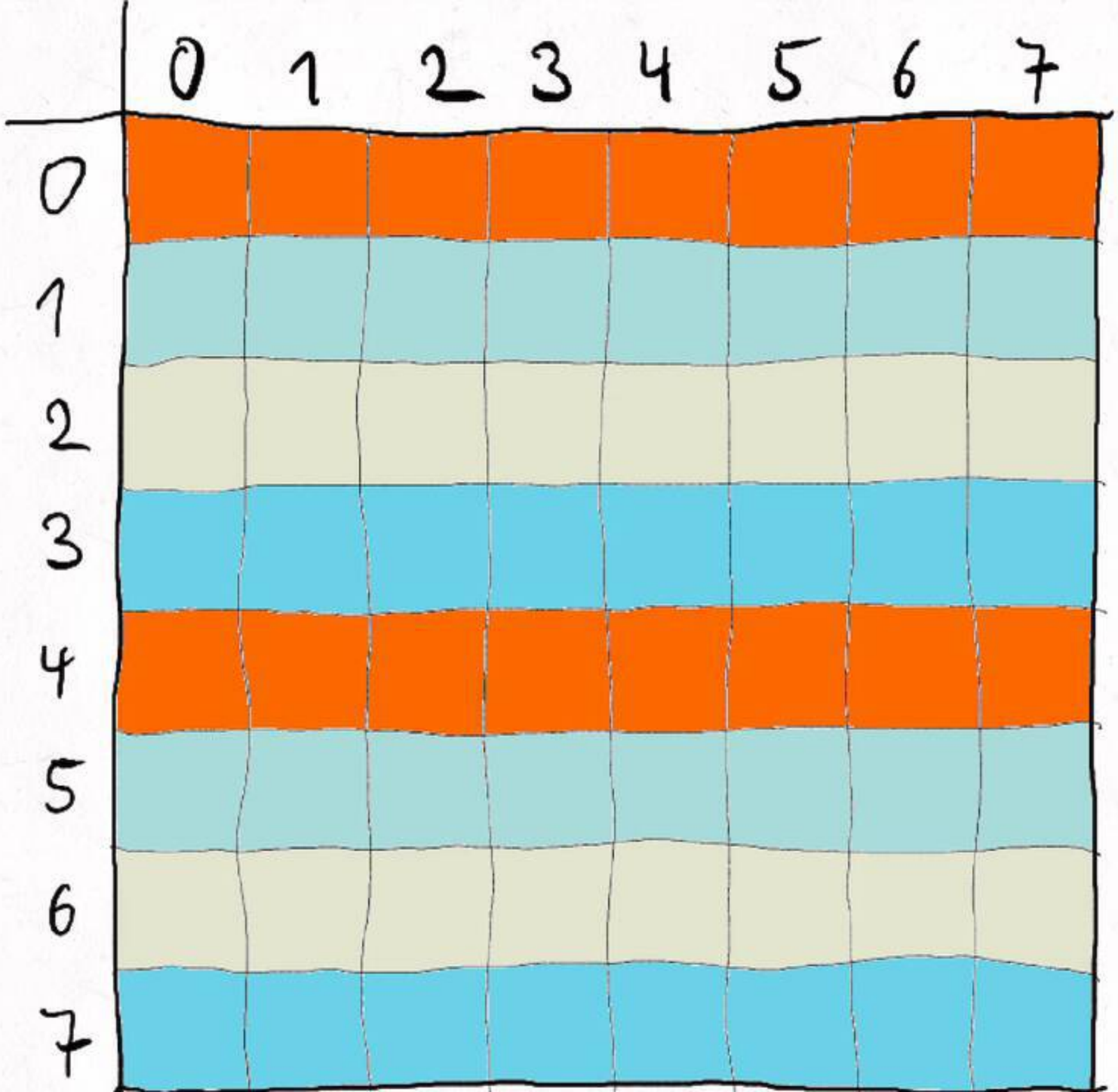


Ändert sich die Intensität, wenn wir nur 4 Threads haben?



Ändert sich die Intensität, wenn wir nur 4 Threads haben?

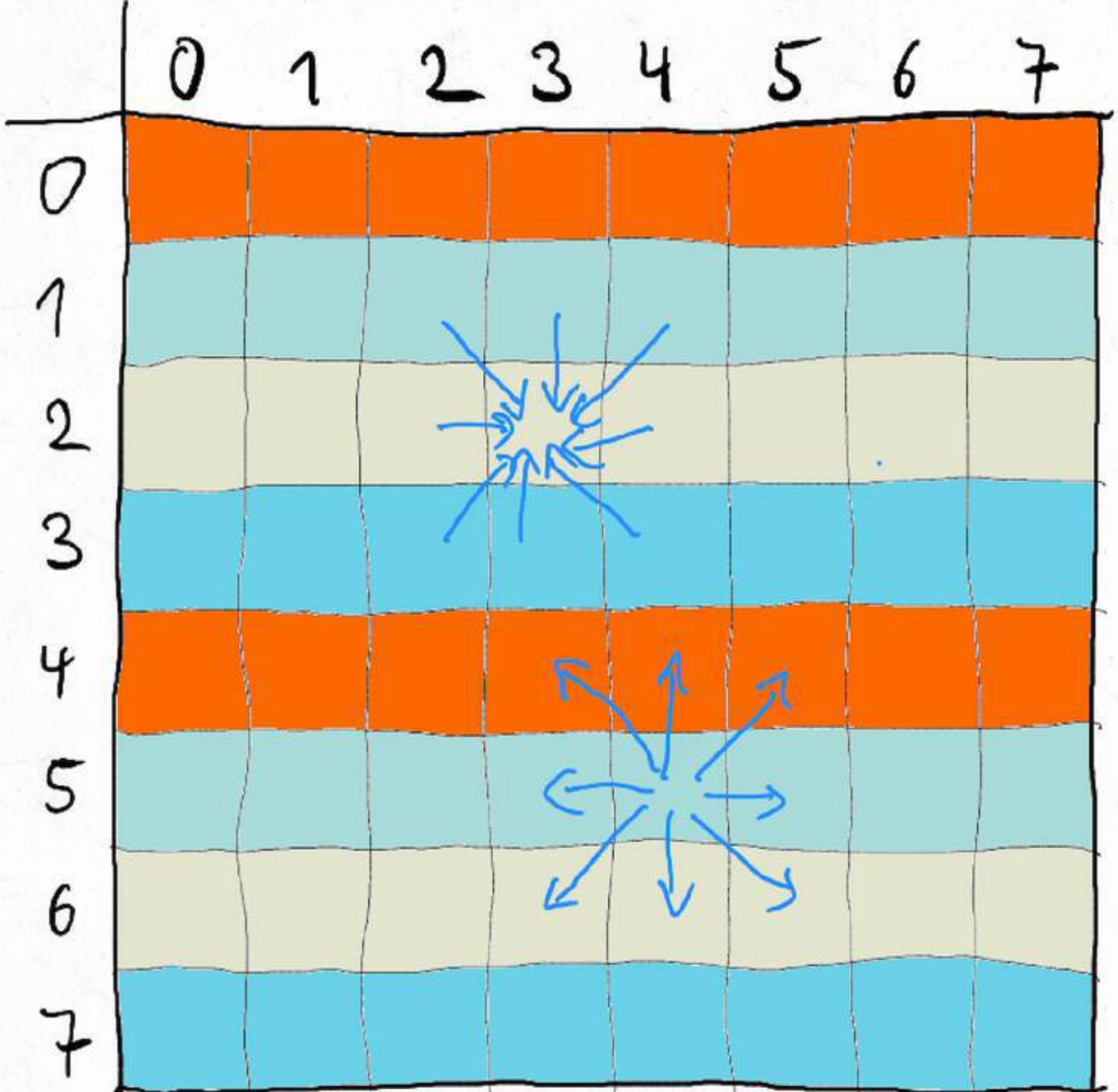
Kommt auf die Zuteilung an!



Ändert sich die Intensität, wenn wir nur 4 Threads haben?

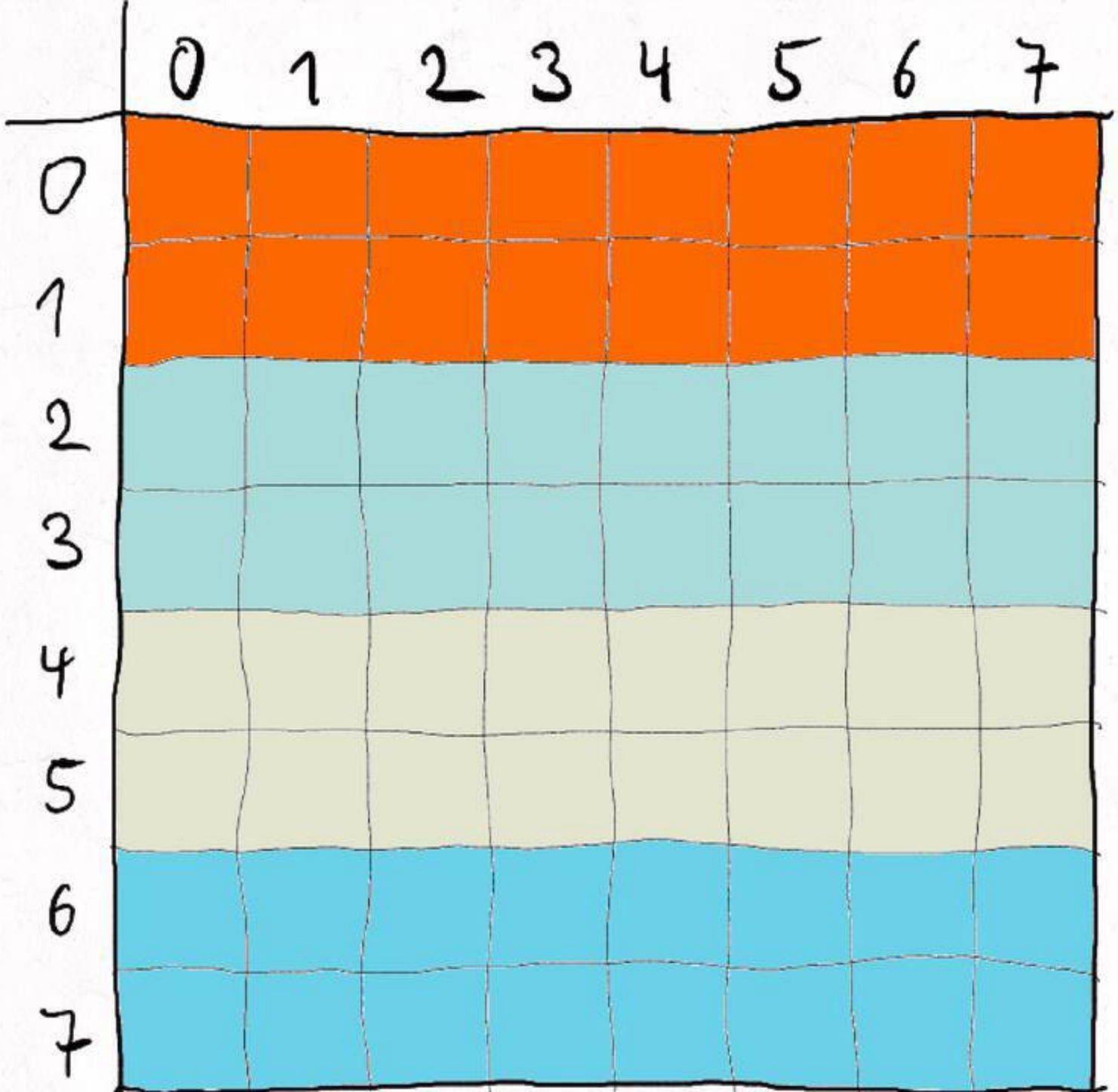
Kommt auf die Zuteilung an!

Möglichkeit 1: wie bisher: 8 Elemte zeilenweise pro Thread
Bei Zeile 4 wieder der erste Thread

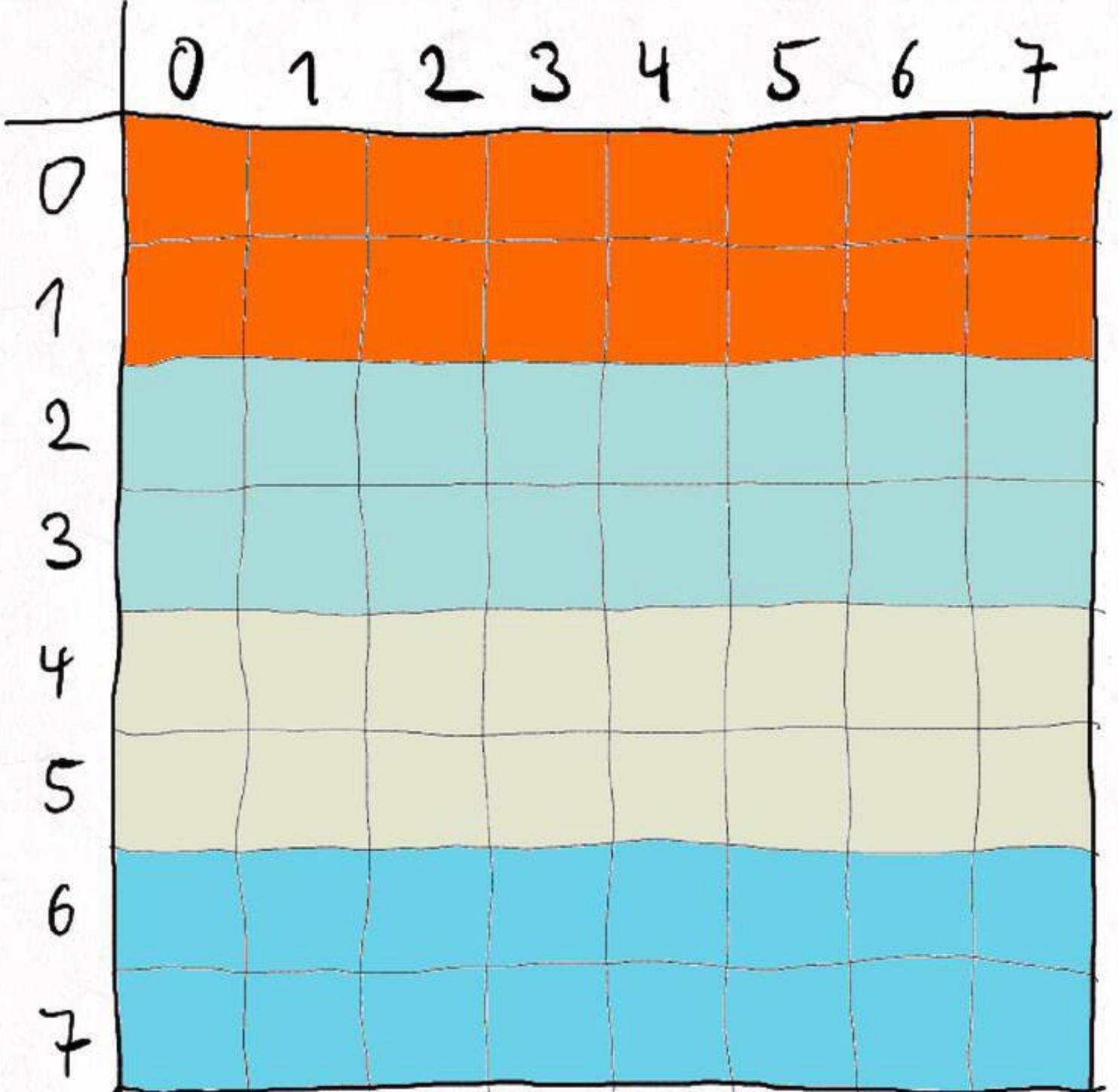


Möglichkeit 1: wie bisher: 8 Elemte zeilenweise pro Thread
Bei Zeile 4 wieder der erste Thread

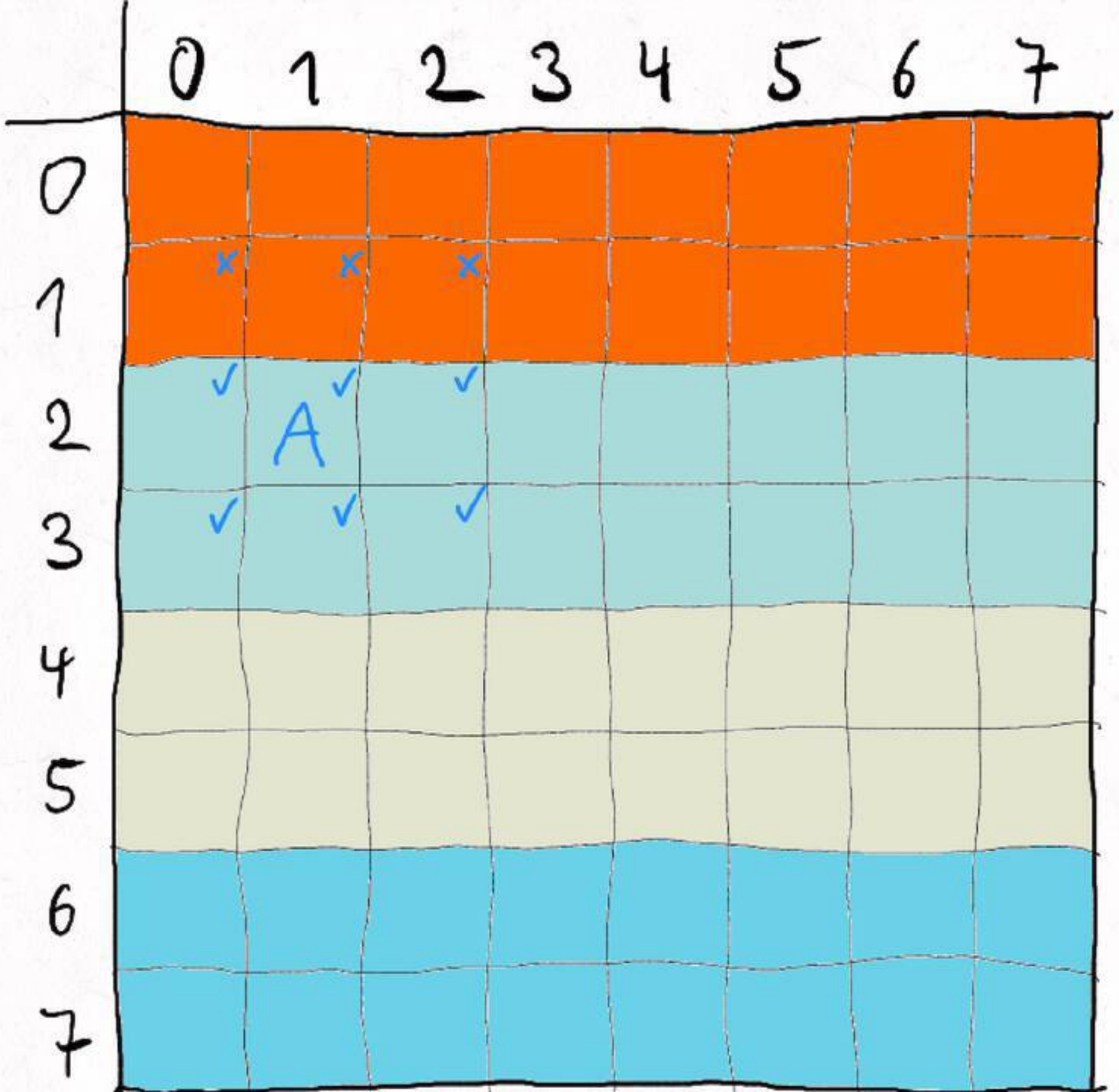
Keine Veränderung
Jedes Element hängt von Daten aus 2 anderen Threads ab und jedes Element wird von anderen Threads benötigt



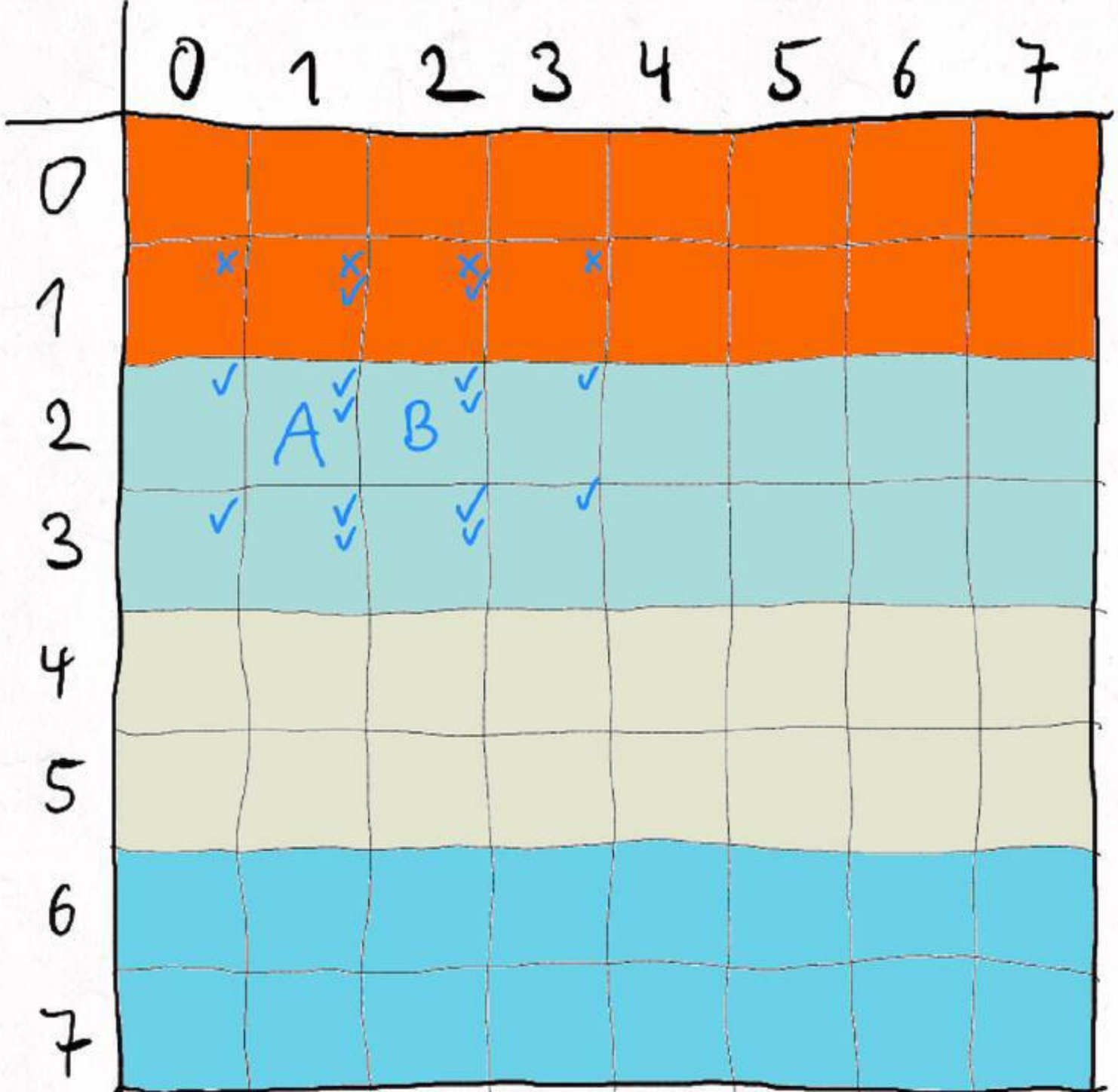
Möglichkeit 2:
Erste N(64)/n(4)=16
Elemente an 1. Thread
usw.



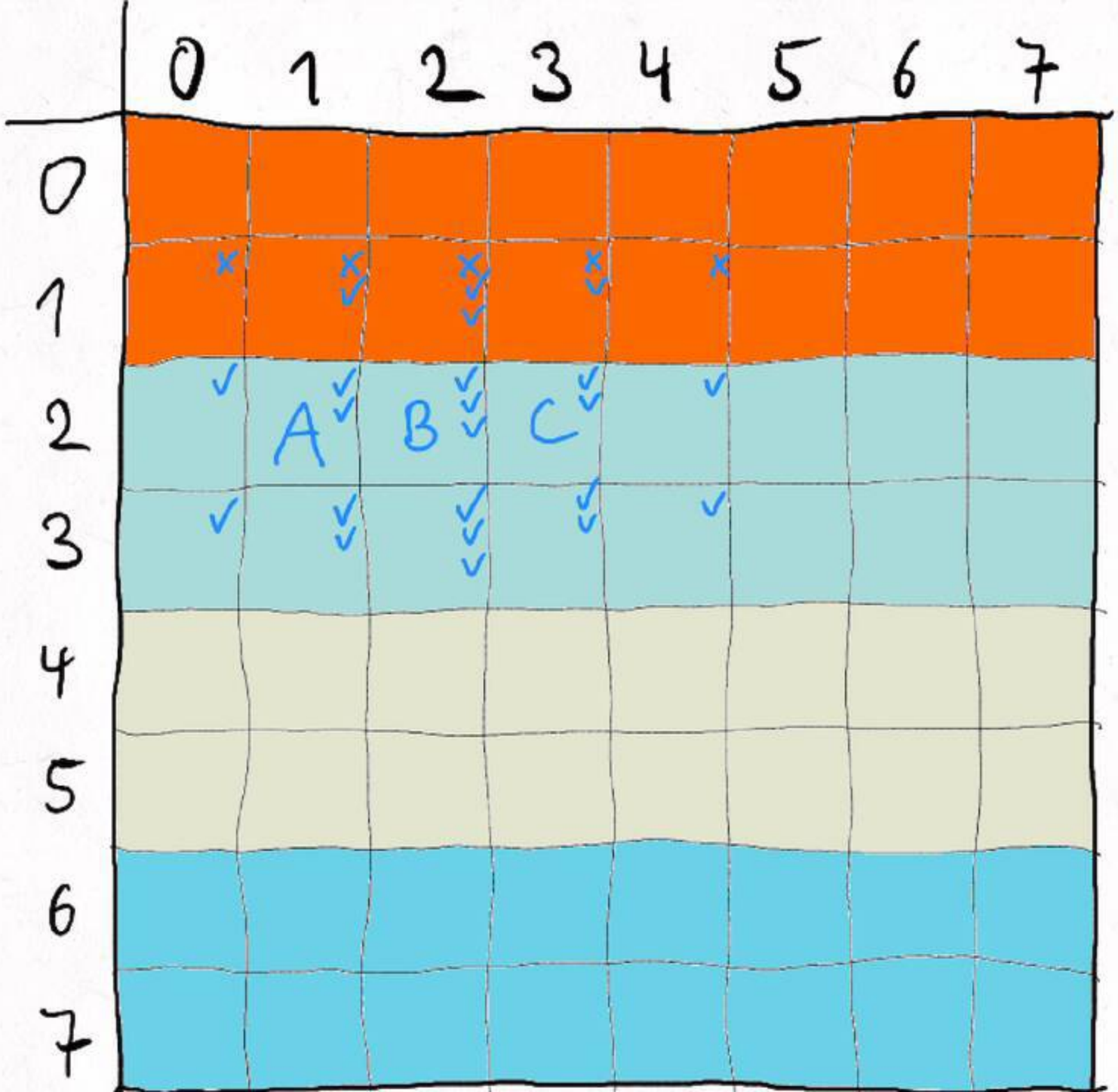
Möglichkeit 2:
Erste N(64)/n(4)=16
Elemente an 1. Thread
usw.
Wir betrachten ab dem
2. Durchgang, d.h. die
Elemente, die dem Thread
zugeordnet sind, sind
schon im Cache



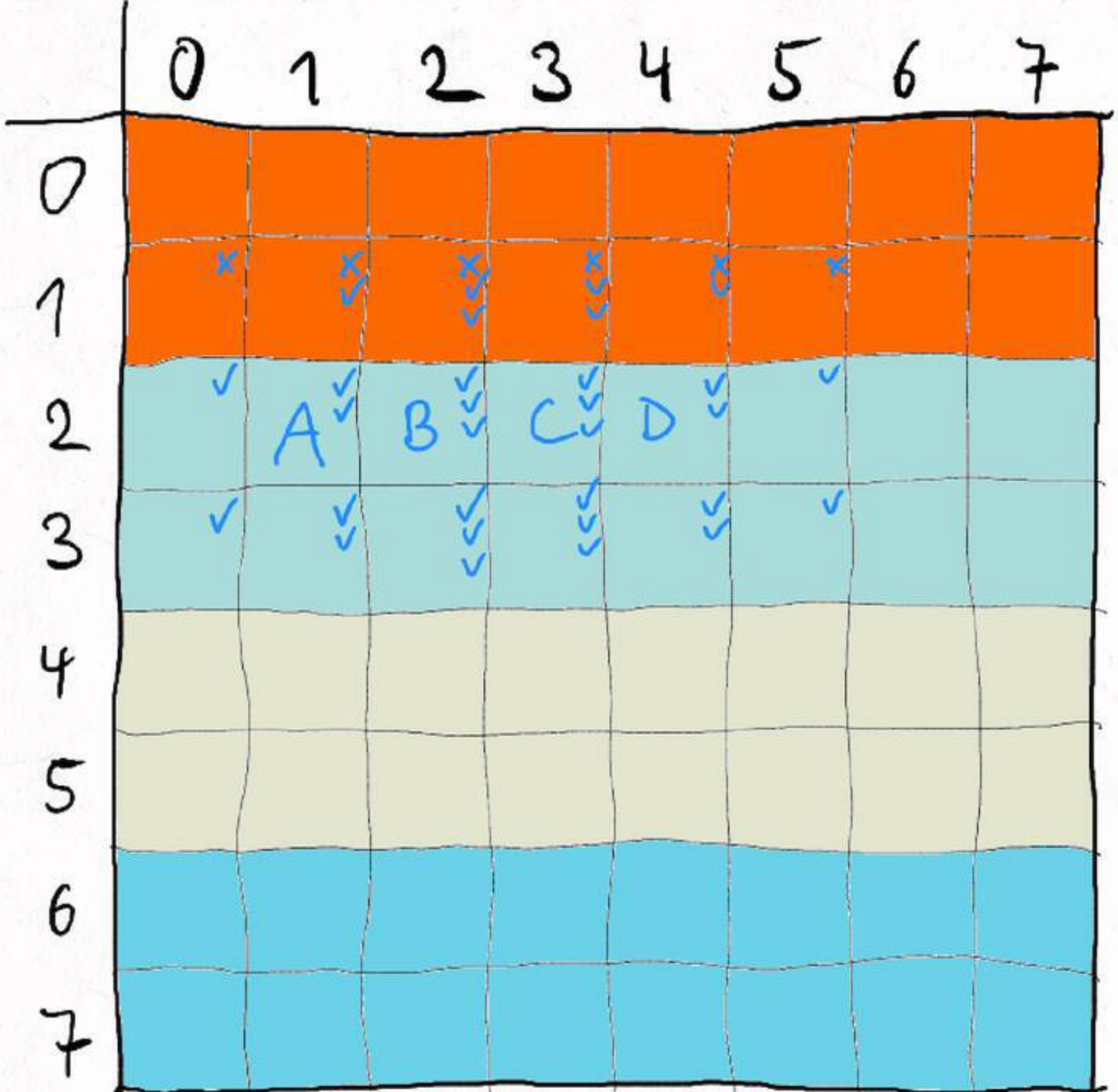
Möglichkeit 2:
Erste N(64)/n(4)=16 Elemente an 1. Thread
usw.
Wir betrachten ab dem 2. Durchgang, d.h. die Elemente, die dem Thread zugeordnet sind, sind schon im Cache:
A: 3



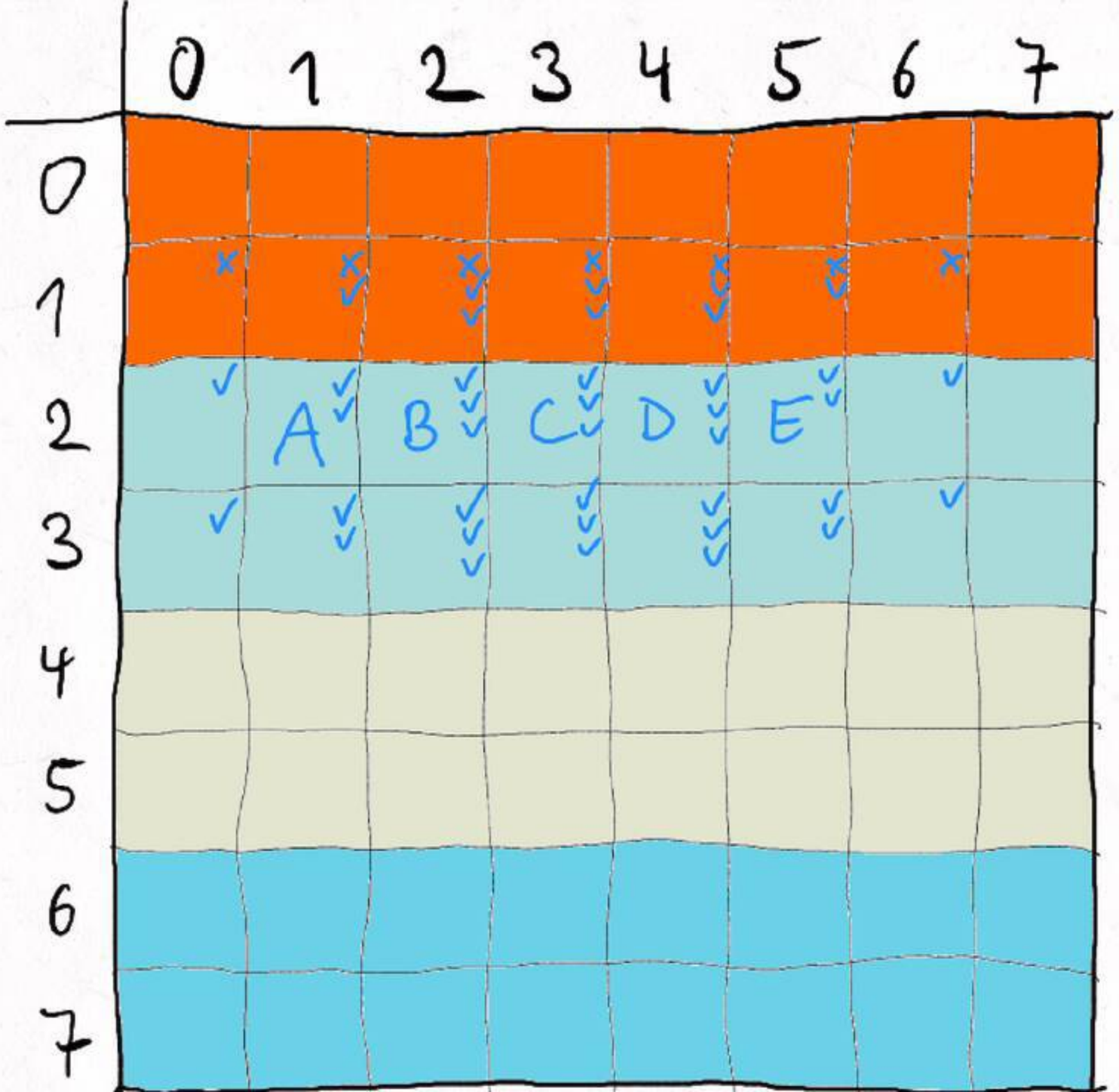
Möglichkeit 2:
Erste N(64)/n(4)=16
Elemente an 1. Thread
usw.
Wir betrachten ab dem
2. Durchgang, d.h. die
Elemente, die dem Thread
zugeordnet sind, sind
schon im Cache:
A: 3 ; B: 1



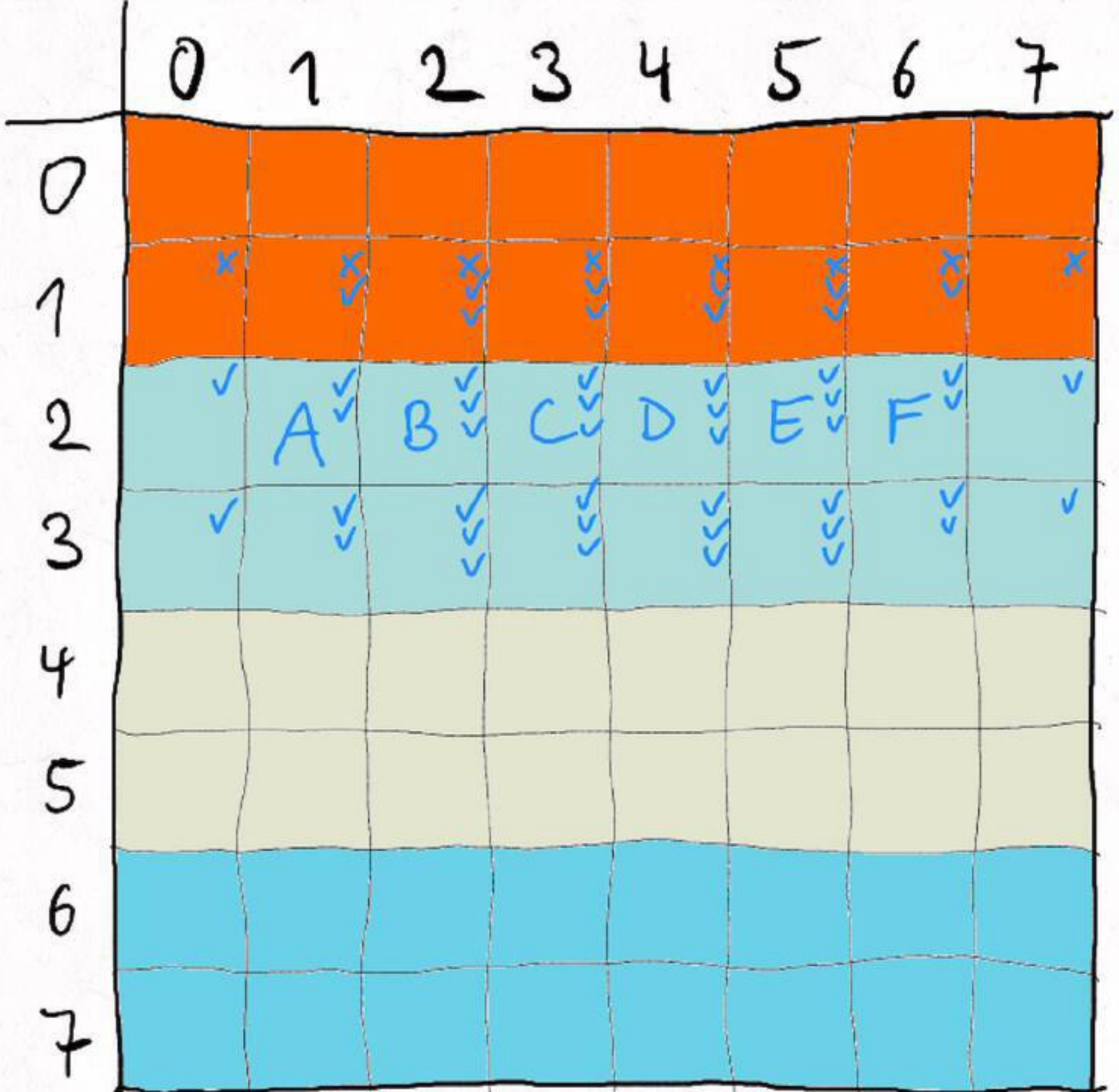
Möglichkeit 2:
Erste N(64)/n(4)=16 Elemente an 1. Thread
usw.
Wir betrachten ab dem 2. Durchgang, d.h. die Elemente, die dem Thread zugeordnet sind, sind schon im Cache:
A: 3 ; B: 1 ; C: 1



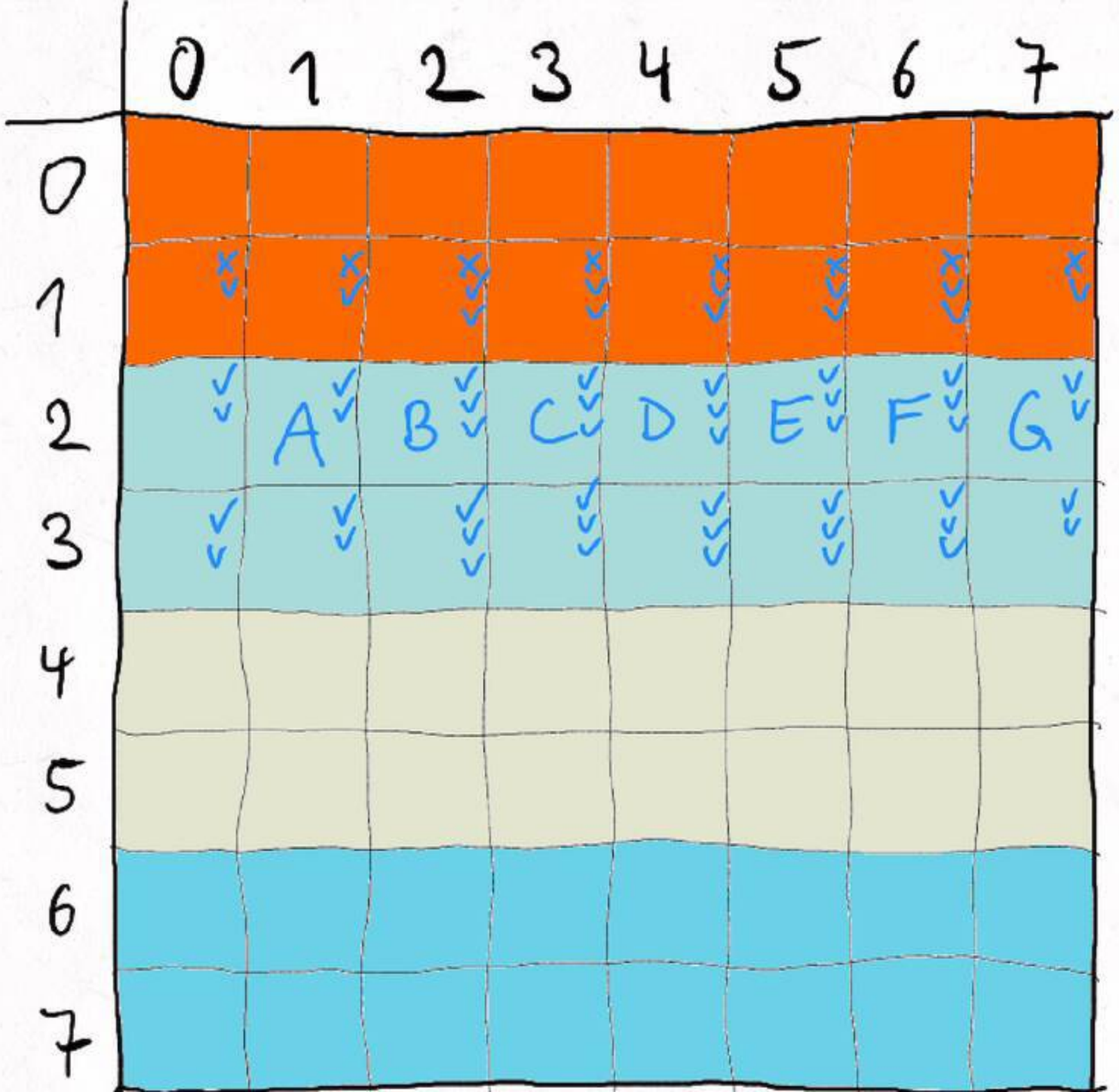
Möglichkeit 2:
Erste N(64)/n(4)=16 Elemente an 1. Thread
usw.
Wir betrachten ab dem 2. Durchgang, d.h. die Elemente, die dem Thread zugeordnet sind, sind schon im Cache:
A: 3 ; B: 1; C: 1; D: 1



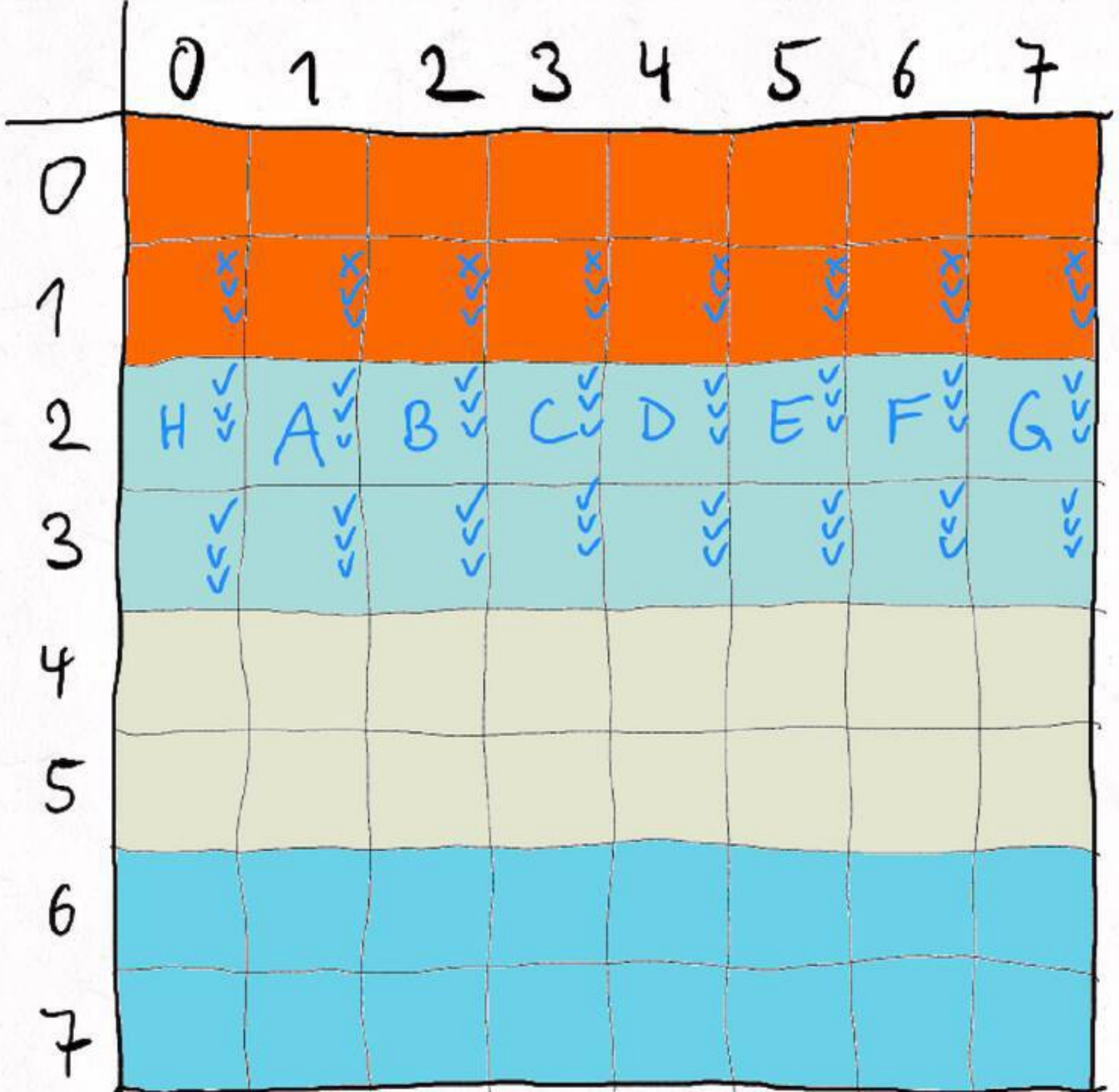
Möglichkeit 2:
Erste N(64)/n(4)=16 Elemente an 1. Thread
usw.
Wir betrachten ab dem 2. Durchgang, d.h. die Elemente, die dem Thread zugeordnet sind, sind schon im Cache:
A: 3 ; B: 1; C: 1; D: 1
E: 1



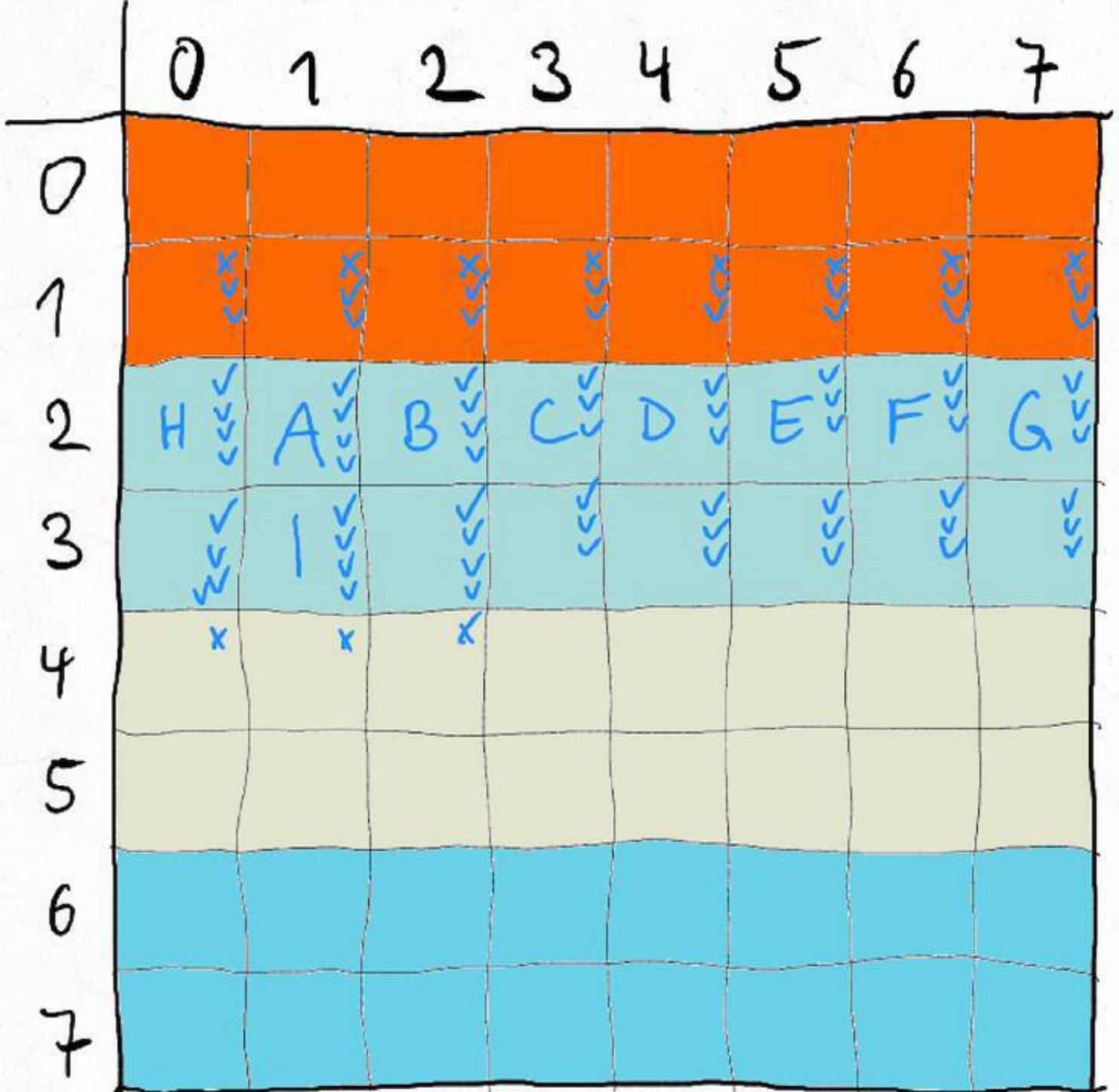
Möglichkeit 2:
Erste N(64)/n(4)=16 Elemente an 1. Thread
usw.
Wir betrachten ab dem 2. Durchgang, d.h. die Elemente, die dem Thread zugeordnet sind, sind schon im Cache:
A: 3 ; B: 1; C: 1; D: 1
E: 1 ; F: 1



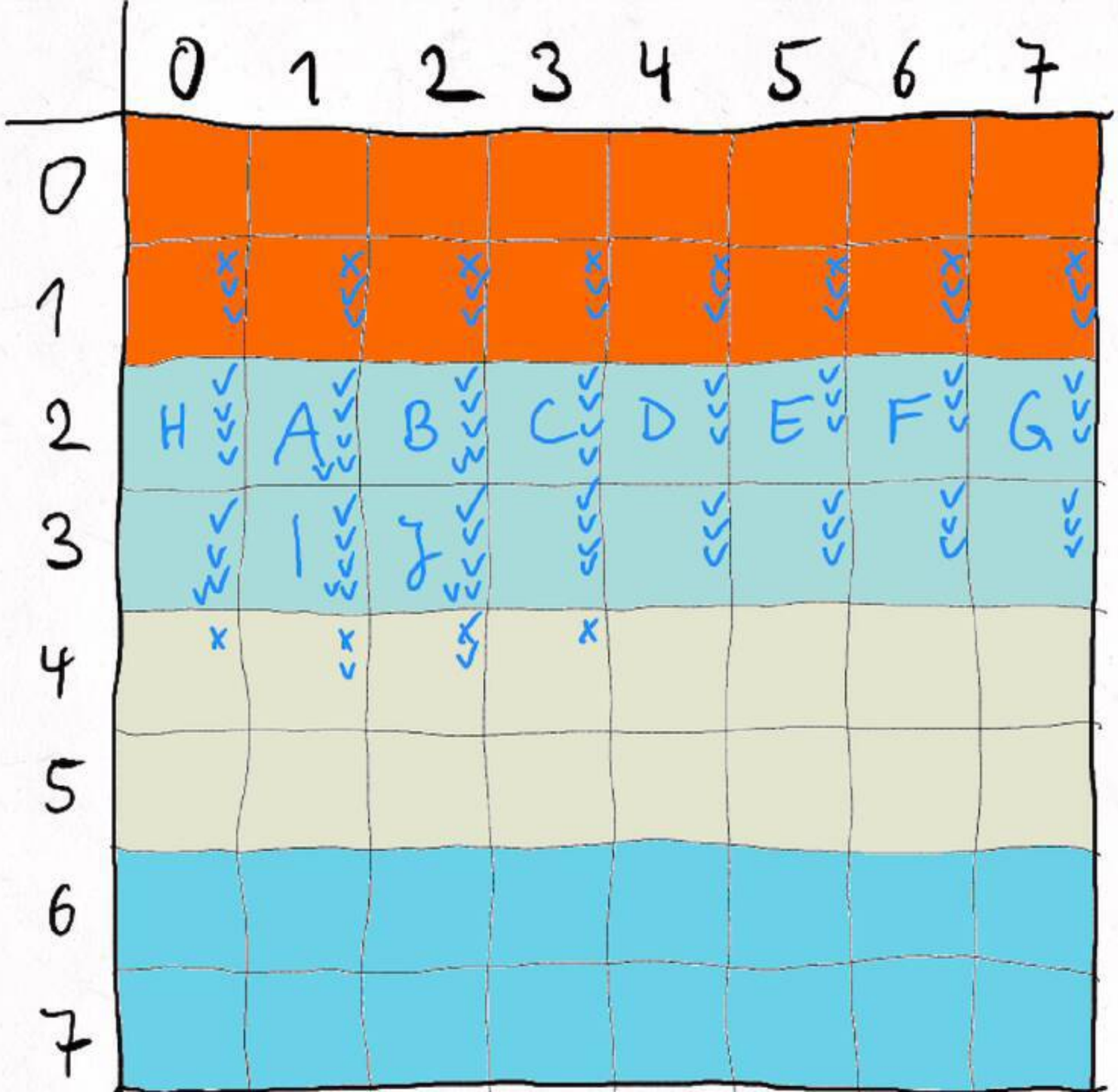
Möglichkeit 2:
Erste N(64)/n(4)=16 Elemente an 1. Thread
usw.
Wir betrachten ab dem 2. Durchgang, d.h. die Elemente, die dem Thread zugeordnet sind, sind schon im Cache:
A: 3 ; B: 1 ; C: 1 ; D: 1
E: 1 ; F: 1 ; G: 0



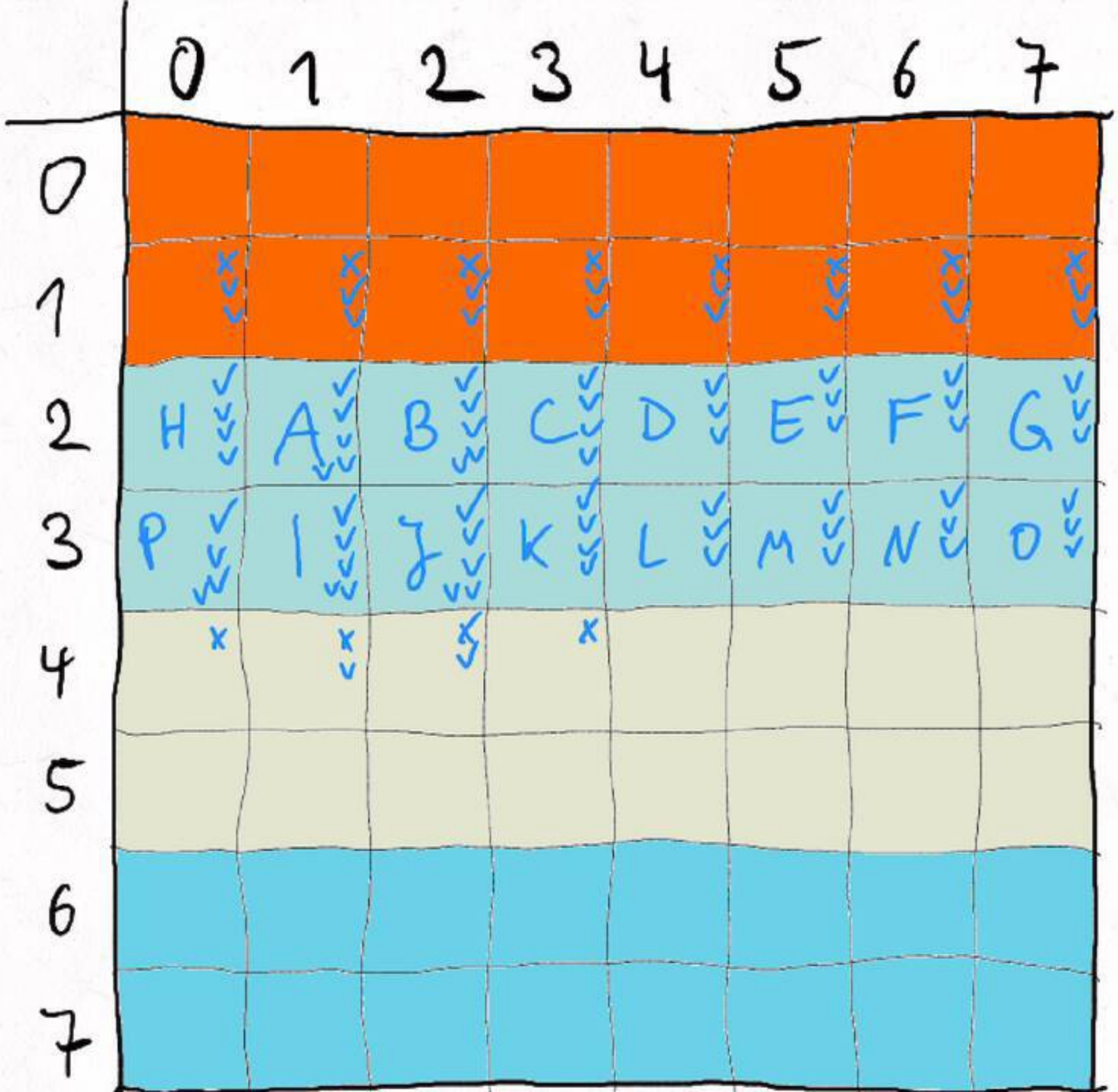
Möglichkeit 2:
Erste N(64)/n(4)=16 Elemente an 1. Thread
usw.
Wir betrachten ab dem 2. Durchgang, d.h. die Elemente, die dem Thread zugeordnet sind, sind schon im Cache:
A: 3 ; B: 1 ; C: 1 ; D: 1
E: 1 ; F: 1 ; G: 0 ; H: 0



Möglichkeit 2:
Erste N(64)/n(4)=16 Elemente an 1. Thread
usw.
Wir betrachten ab dem 2. Durchgang, d.h. die Elemente, die dem Thread zugeordnet sind, sind schon im Cache:
A: 3 ; B: 1 ; C: 1 ; D: 1
E: 1 ; F: 1 ; G: 0 ; H: 0
I: 3

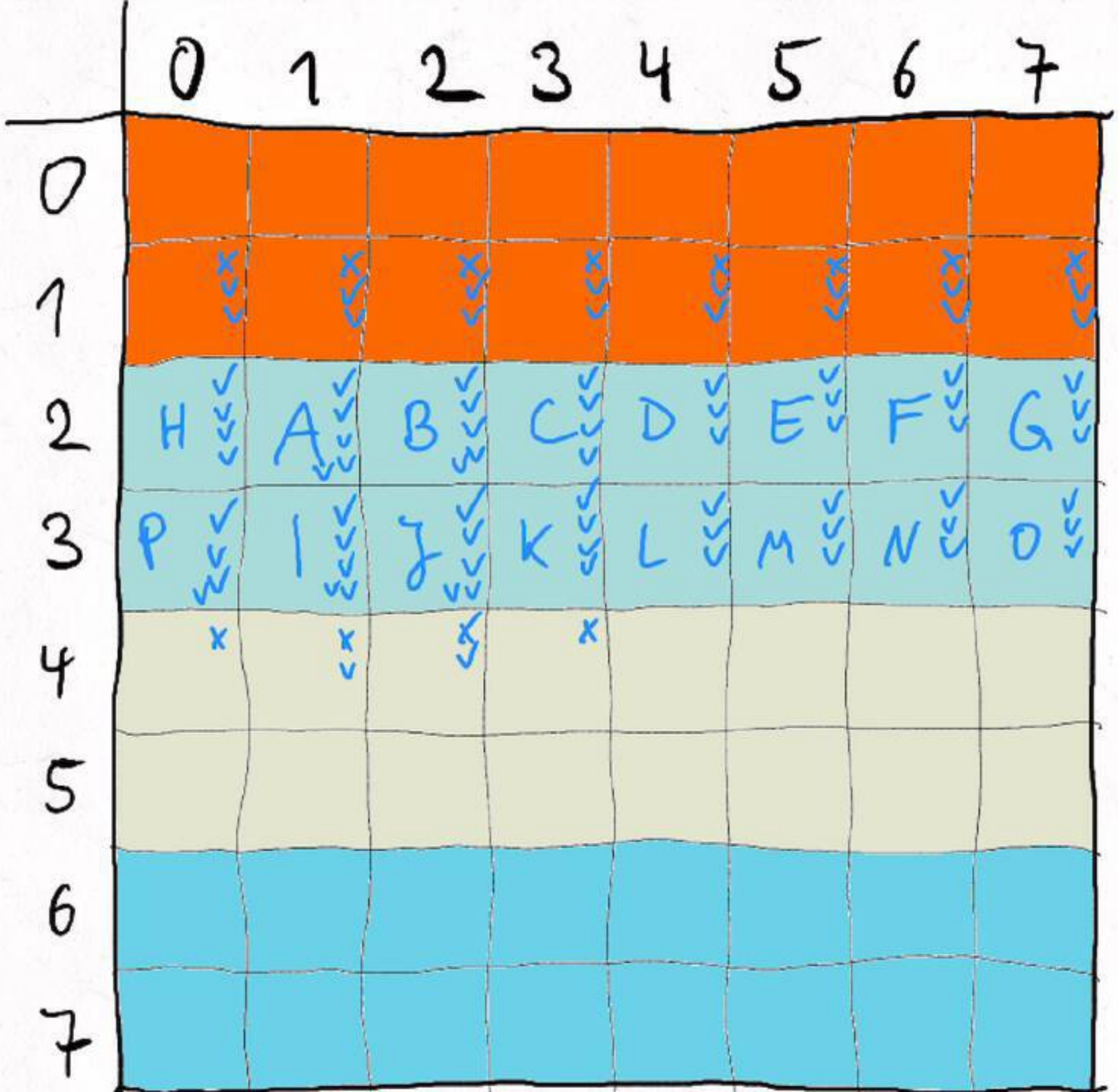


Möglichkeit 2:
Erste N(64)/n(4)=16 Elemente an 1. Thread
usw.
Wir betrachten ab dem 2. Durchgang, d.h. die Elemente, die dem Thread zugeordnet sind, sind schon im Cache:
A: 3 ; B: 1 ; C: 1 ; D: 1
E: 1 ; F: 1 ; G: 0 ; H: 0
I: 3 ; J: 1



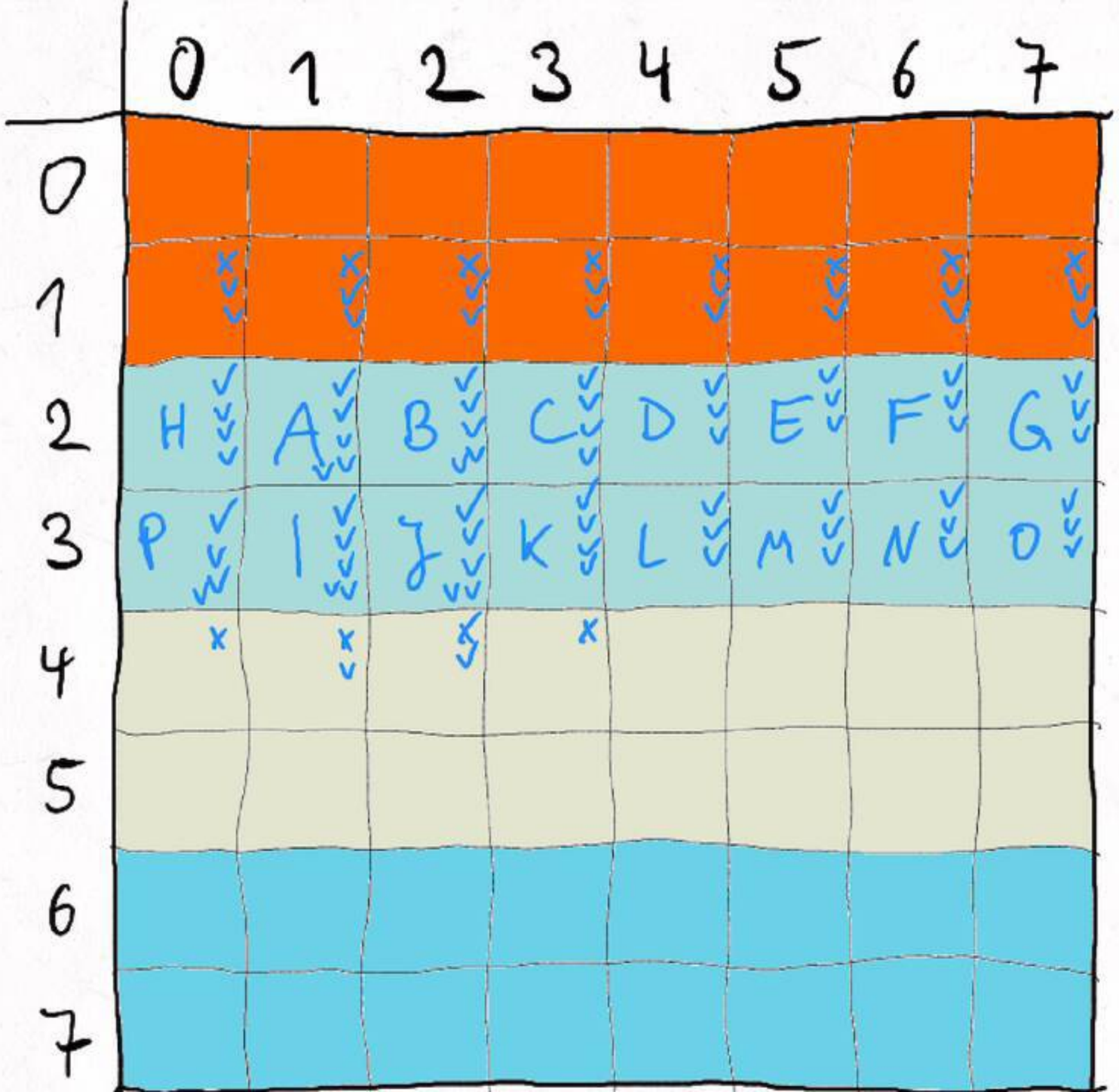
Möglichkeit 2:
Erste N(64)/n(4)=16 Elemente an 1. Thread usw.
Wir betrachten ab dem 2. Durchgang, d.h. die Elemente, die dem Thread zugeordnet sind, sind schon im Cache:

A: 3 ; B: 1 ; C: 1 ; D: 1
E: 1 ; F: 1 ; G: 0 ; H: 0
I: 3 ; J: 1 ; K: 1 ; L: 1
M: 1 ; N: 1 ; O: 0 ; P: 0
Σ: 16 Reads



Möglichkeit 2:
Erste N(64)/n(4)=16
Elemente an 1. Thread
usw.
Wir betrachten ab dem
2. Durchgang

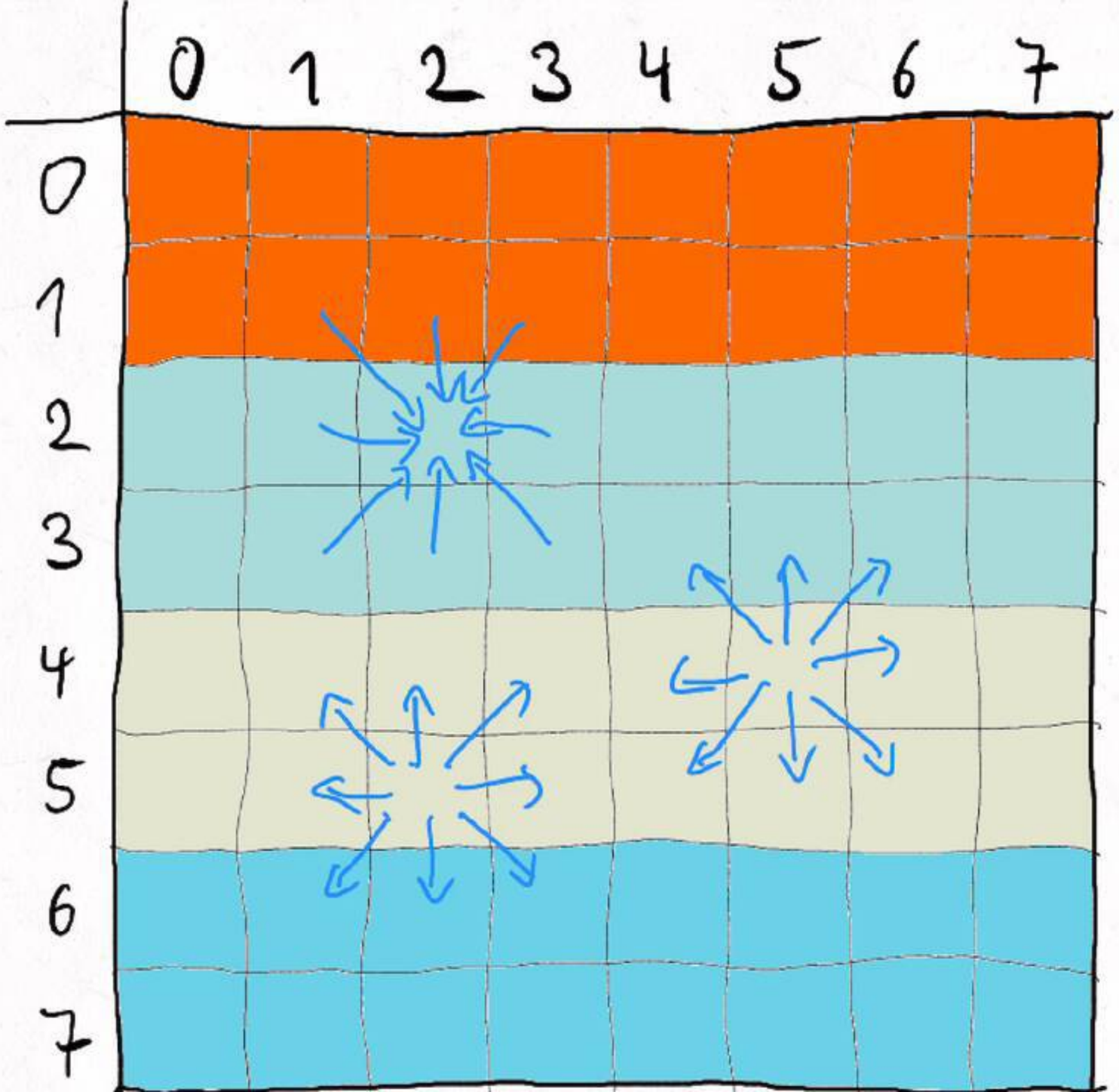
Wieder 16 Reads
pro Durchgang
und Thread?



Möglichkeit 2:
Erste N(64)/n(4)=16
Elemente an 1. Thread
usw.
Wir betrachten ab dem
2. Durchgang

Wieder 16 Reads
pro Durchgang
und Thread?

Ja, aber nur noch
4 Threads ...
... also 4x16=64 statt 128!



Möglichkeit 2:
Erste N(64)/n(4)=16
Elemente an 1. Thread
usw.
Wir betrachten ab dem
2. Durchgang.

- Jedes Element braucht nur noch Daten aus 1 anderen Thread :)
- Jedes Element wird immer noch von 1 anderen Thread gebraucht ...
... immer noch 64 Writes :(

Wie compute-intensive ist Möglichkeit 2?

$4 \times 16 = 64$ Reads pro Durchgang

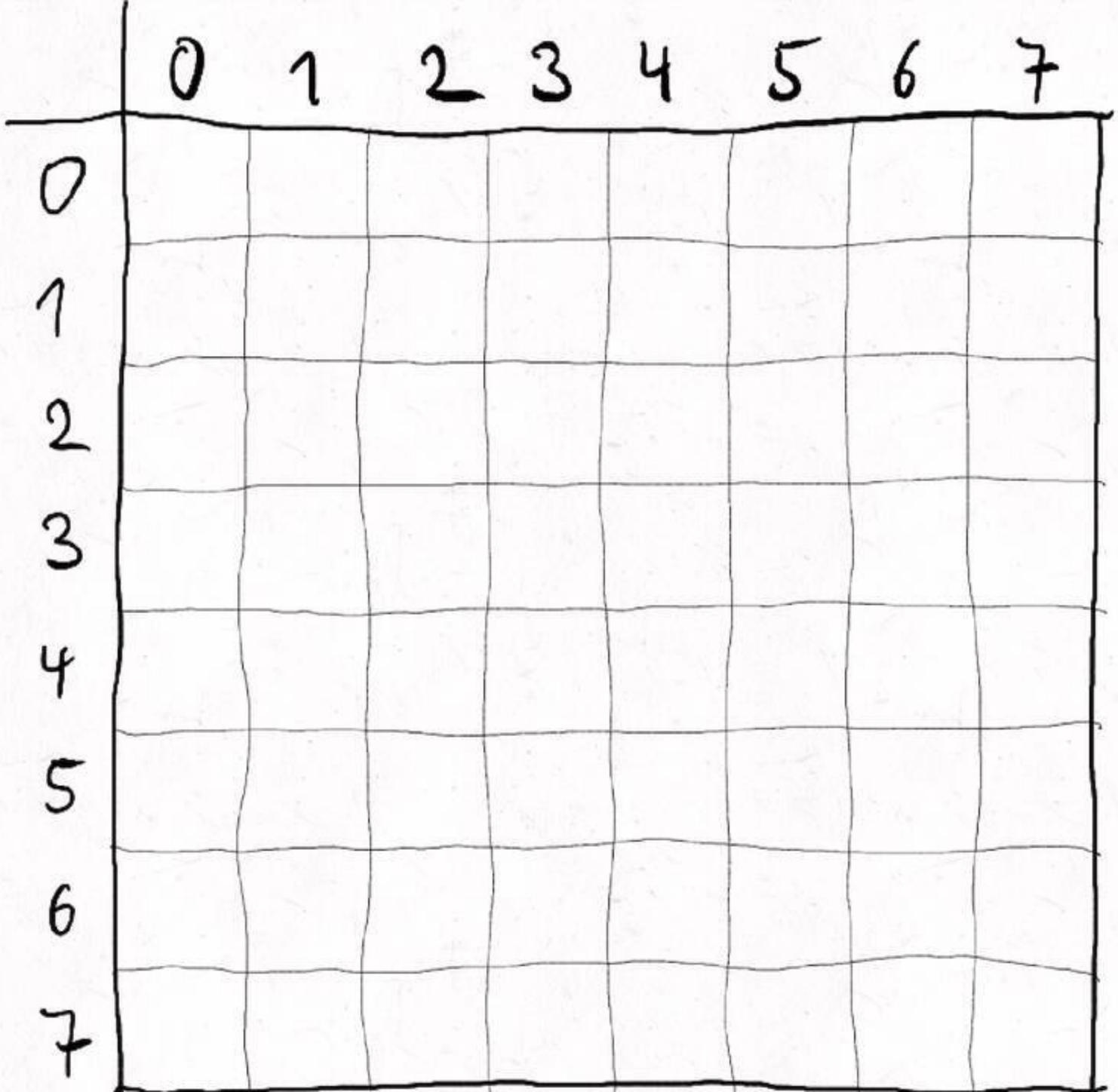
$+ \quad 8 \times 8 = 64$ Writes pro Durchgang

jeder Wert wird von einem anderen Thread ausgelesen – muss also in den Speicher

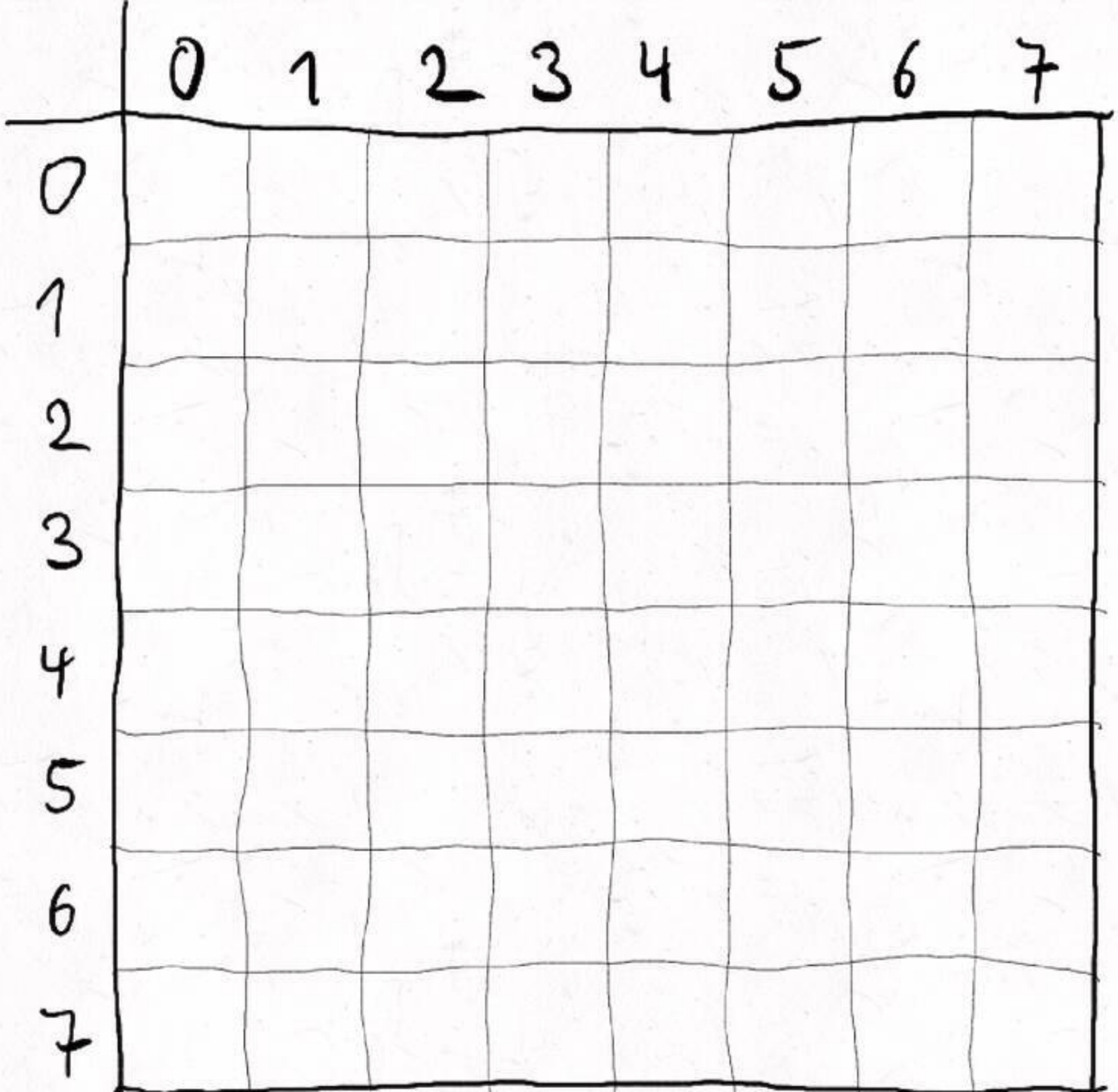
128×4 Bytes (Single Prec. Float) $= 512$ Bytes

Floating Point Ops: 64 Felder $\times$ (1 Multiplikation + 8 Additionen) $= 576$

Intensität: $576 / 512 = 1{,}125$ (+50% zu Möglichkeit 1)

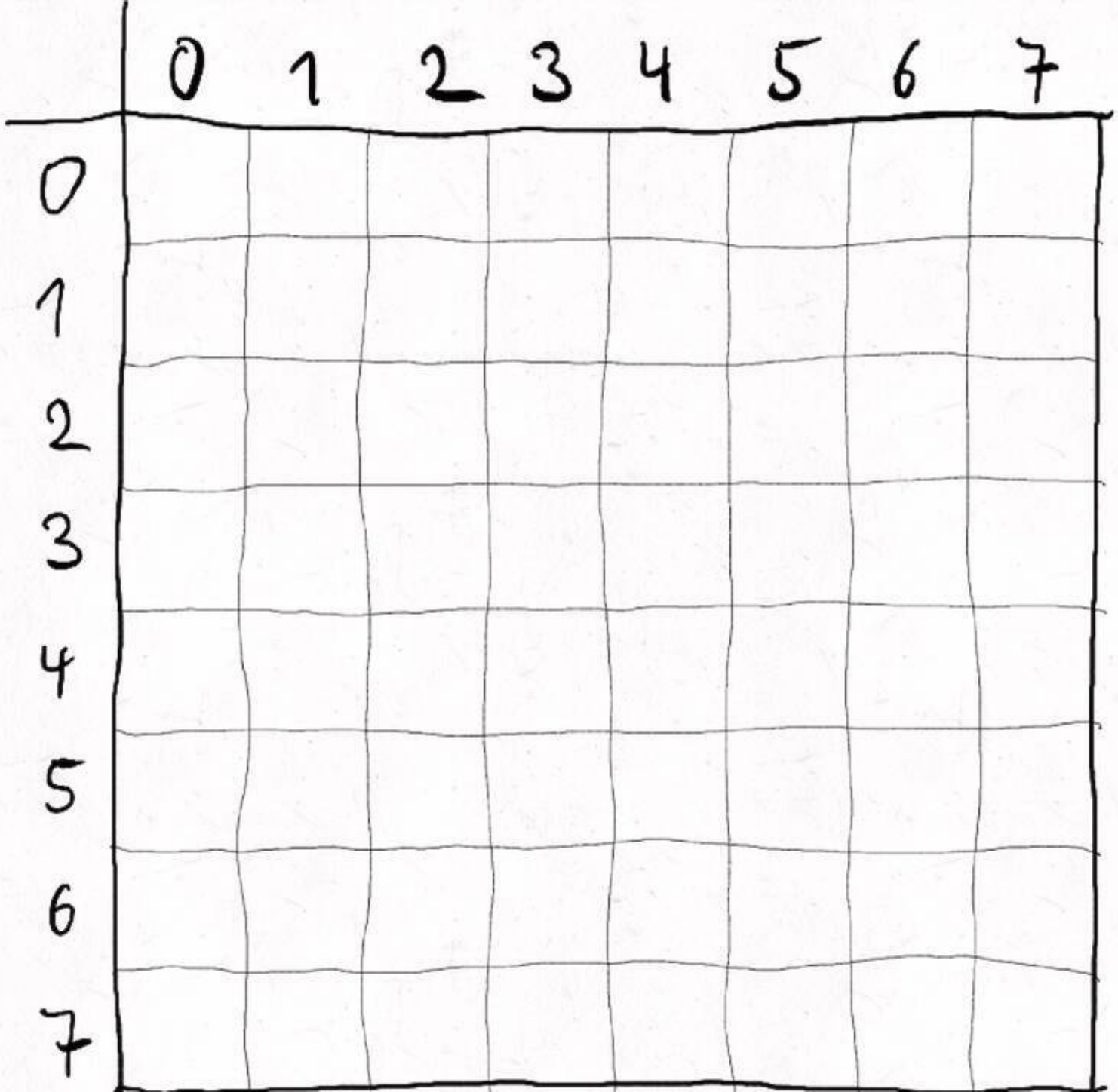


Gibt es eine Aufteilung auf 4 Threads mit noch höherer Intensität als Möglichkeit 2?



Gibt es eine Aufteilung auf 4 Threads mit noch höherer Intensität als Möglichkeit 2?

Ja!



Gibt es eine Aufteilung auf 4 Threads mit noch höherer Intensität als Möglichkeit 2?

Ja!

Ideen?