Toward GPU Accelerated Data Stream Processing

Marcus Pinnecke, David Broneske and Gunter Saake
University of Magdeburg, Germany

May 27, 2015

Background and Motivation

Fundamentals, Windowing, GPU Acceleration in DBMS/SPS

Data Stream Processing
Application requirements

Examples

- **System Monitoring and Fraud Prevention** — Log files about load, network activity, storage
- **Social Media** — Identify topics of interest online, such as *top-k* hash tags on Twitter
- ...

Requirements

- Real-time response
- Continuous processing and analysis
- High-volume data, potentially infinite
- High-velocity data (many changes)

Data Stream Processing

Data Stream Processing
Processing Model and Windowing

Infinite streams of data, but…

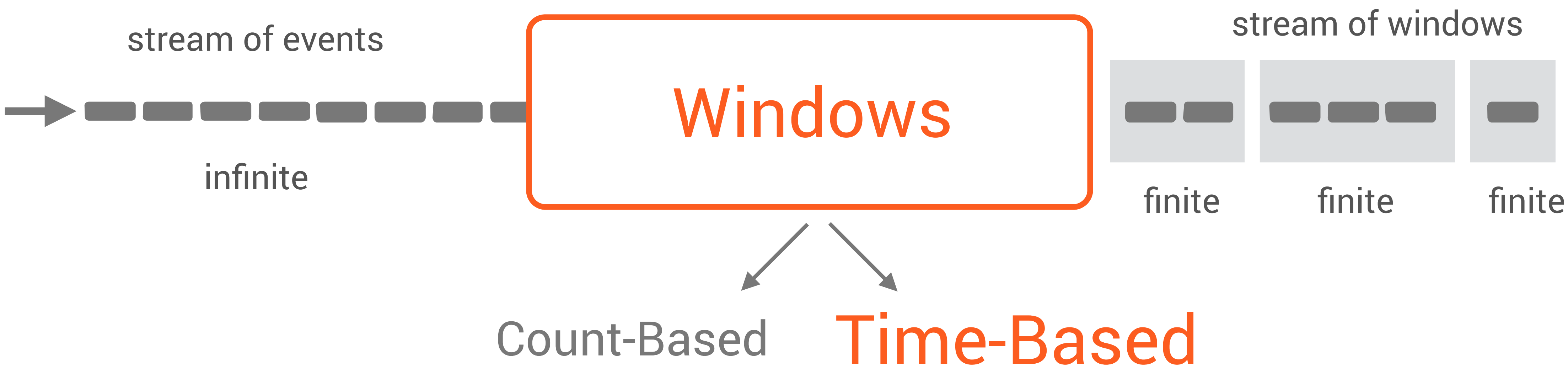
- **Limited** main **memory** and
- Only **sequential access**

Solutions

- Reduction of data amount (e.g., sampling) or
- Buffering (**windowing**)

Data Stream Processing

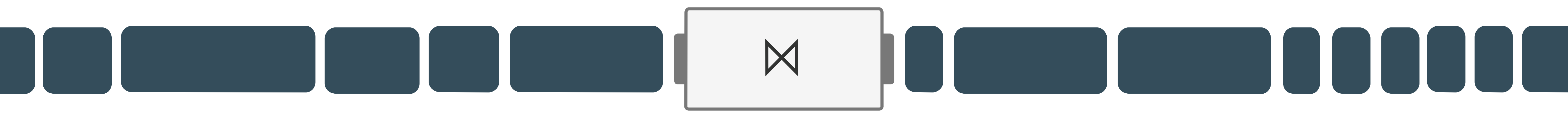
Processing Model and Windowing



- More common for real applications
- Variable number of events per window
- Problematic due to limited GPU memory

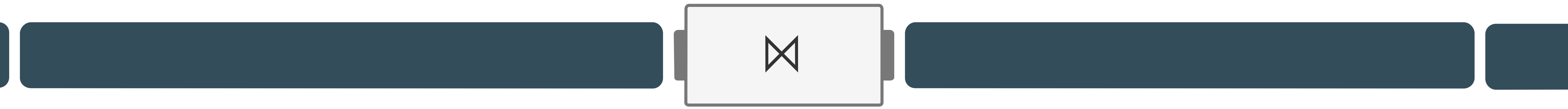
Data Stream Processing
Bottleneck — Example Join Algorithm

- Number of join candidates depends on number of events inside window



Data Stream Processing
Bottleneck — Example Join Algorithm

- Number of join candidates depends on number of events inside window
- Many events in the same instant for time-based windows
  - Decrease of throughput



Data Stream Processing
Bottleneck — Back Pressure

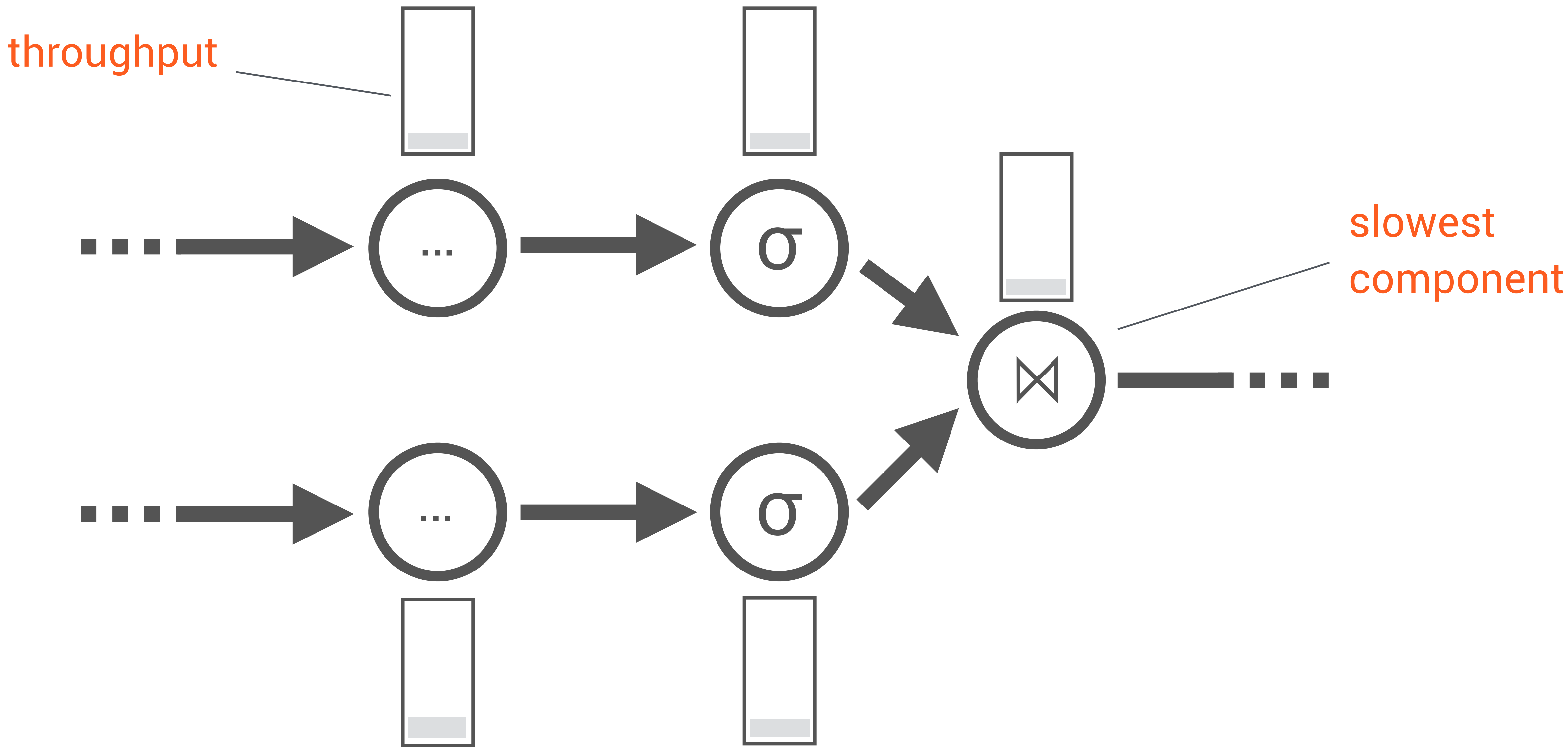
Data flow systems (e.g., stream processing) suffer of **back pressure**

Back pressure

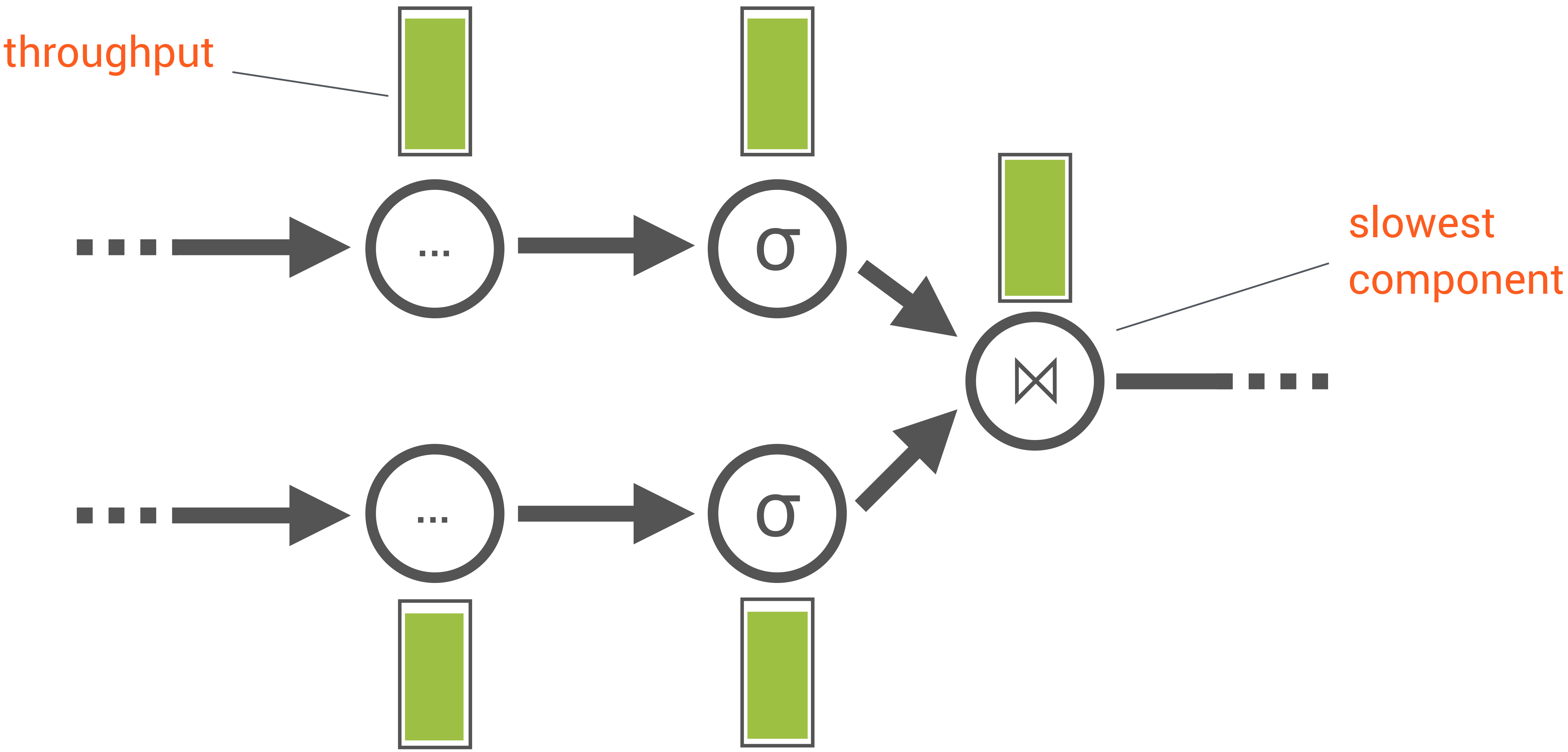
- Upwards-propagated decrease of throughput
- To the level of the slowest component

Results is need for load shedding.

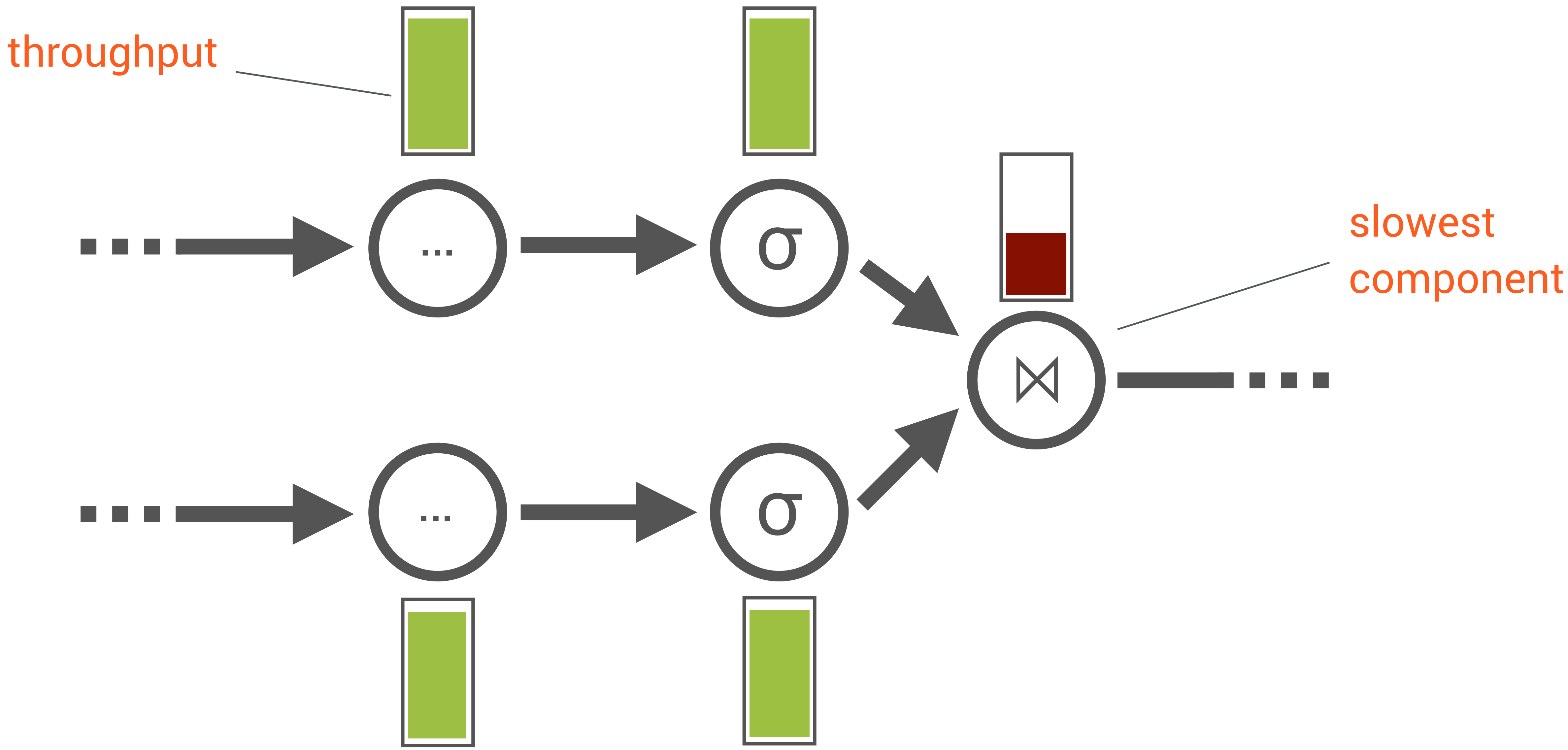
Data Stream Processing
Bottleneck



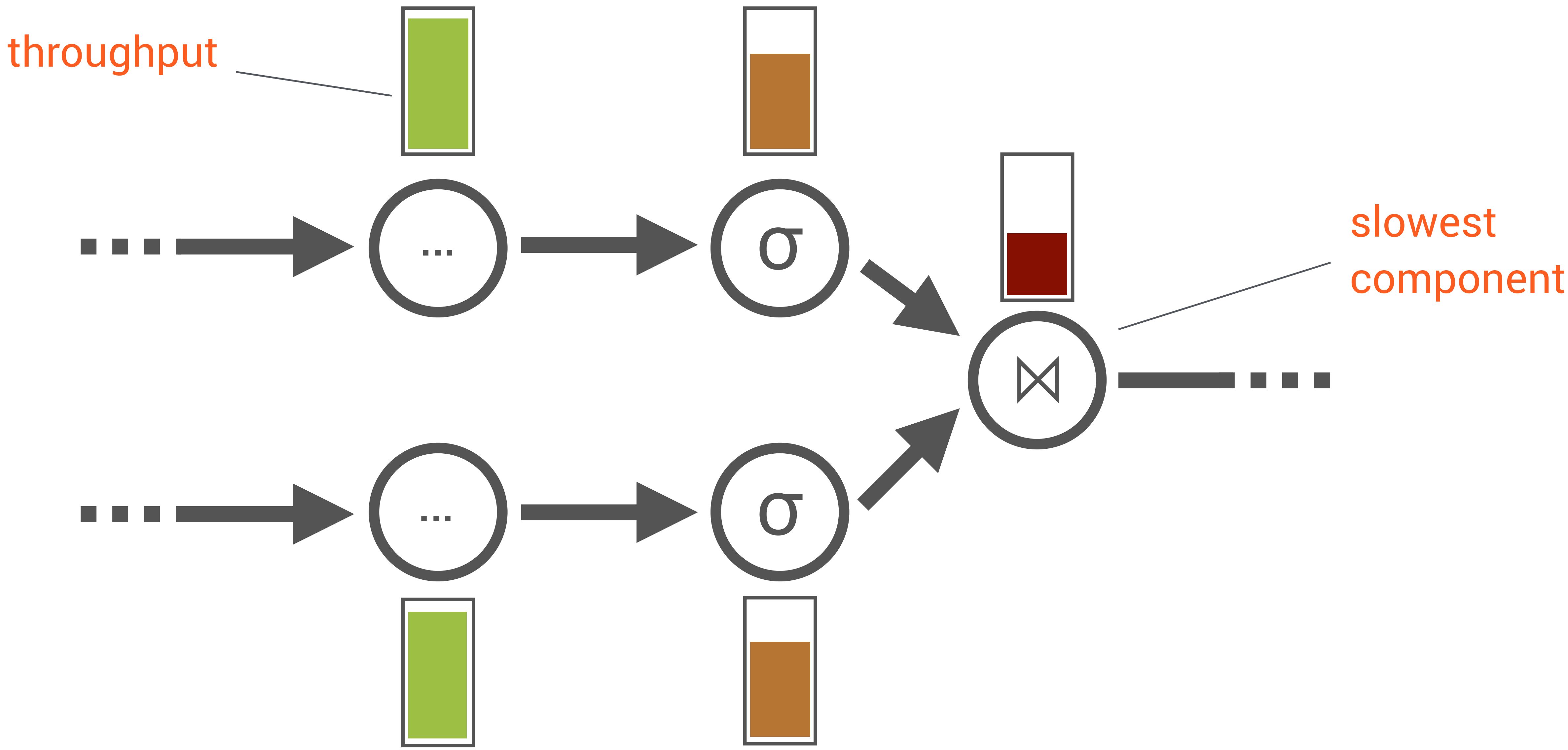
Data Stream Processing
Bottleneck



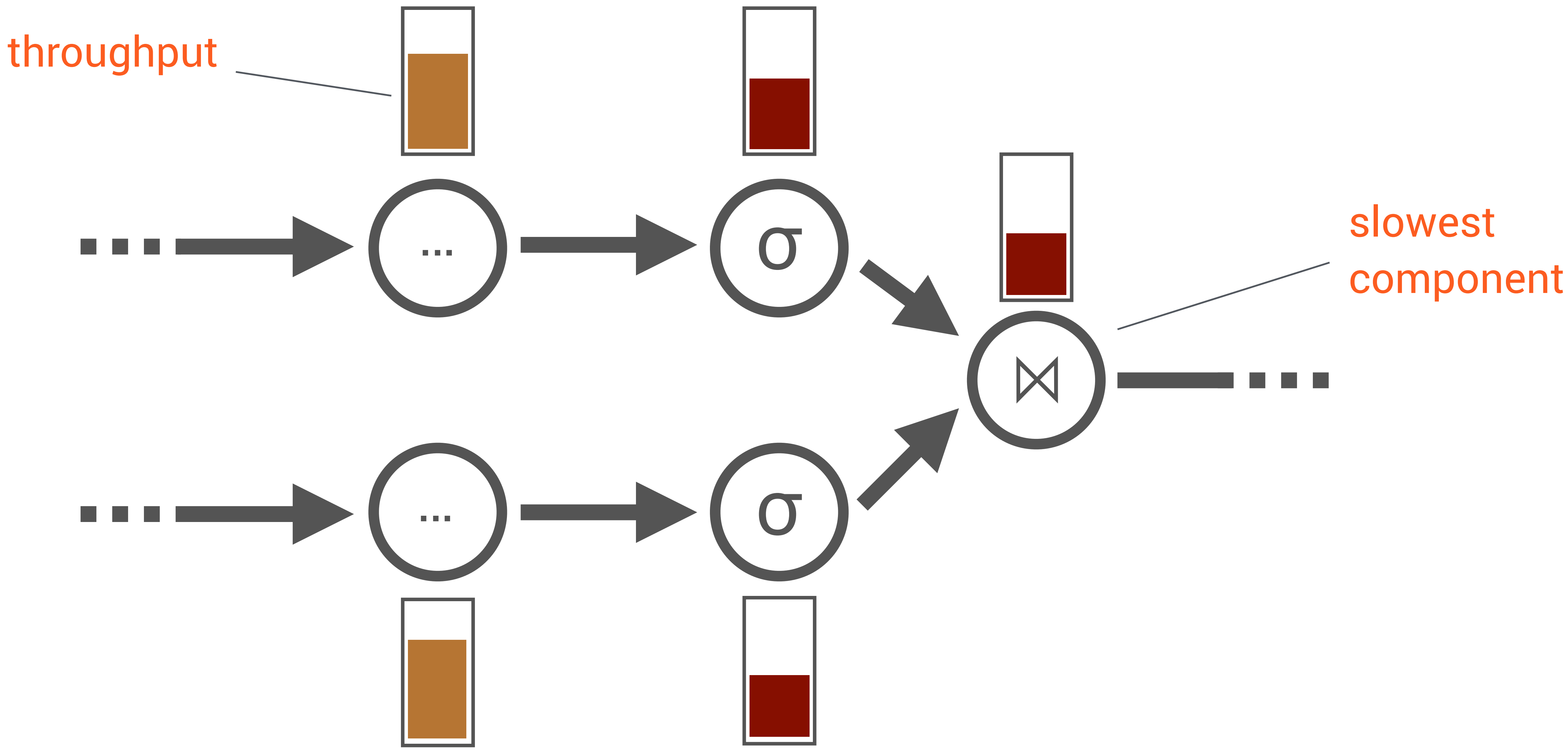
Data Stream Processing
Bottleneck



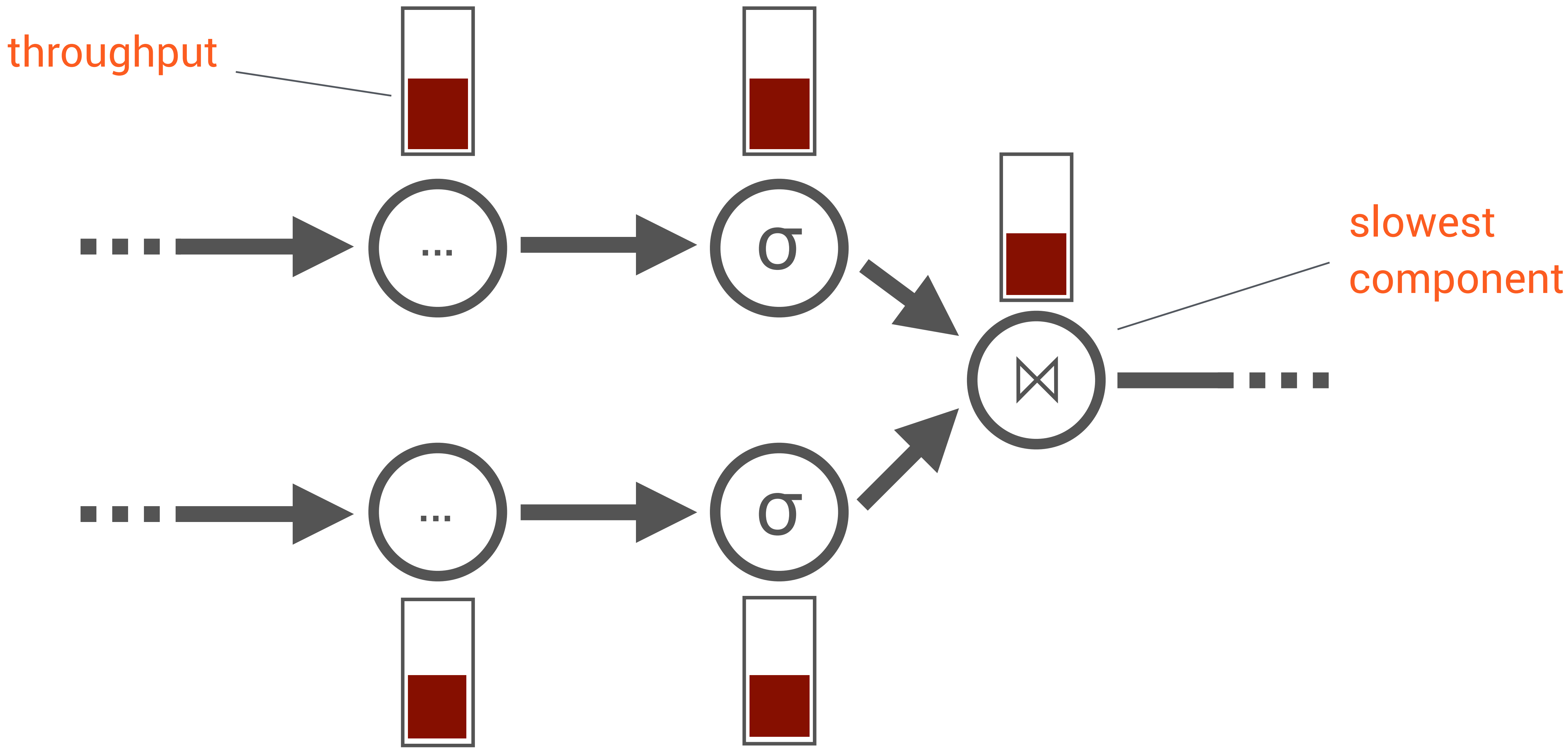
Data Stream Processing
Bottleneck



Data Stream Processing
Bottleneck

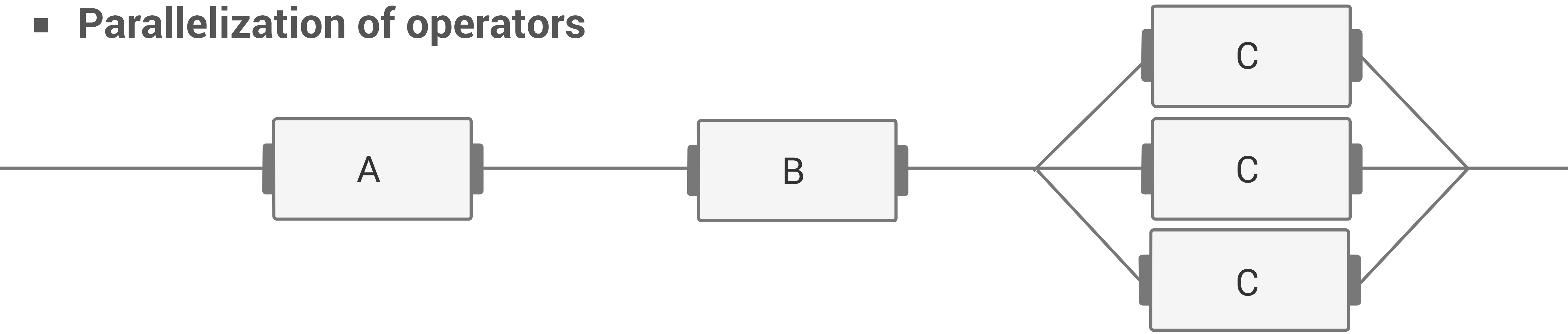


Data Stream Processing
Bottleneck

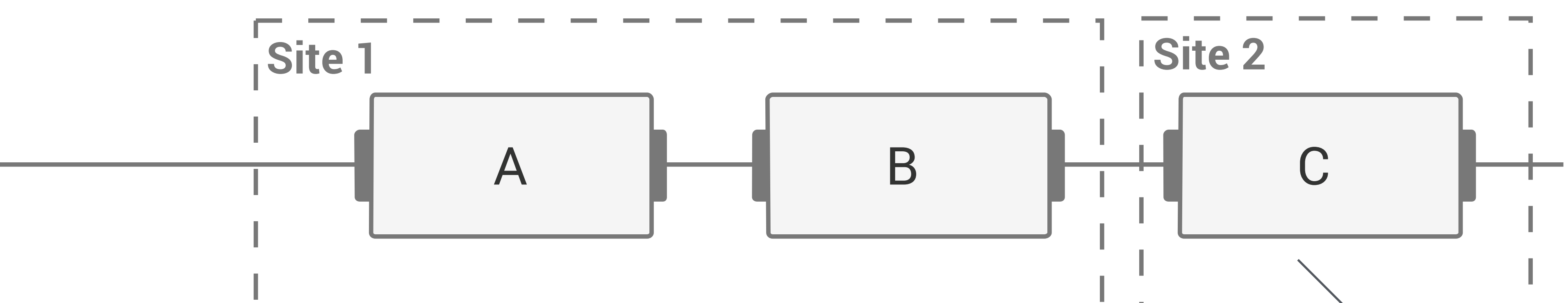


Data Stream Processing
Bottleneck – Solutions

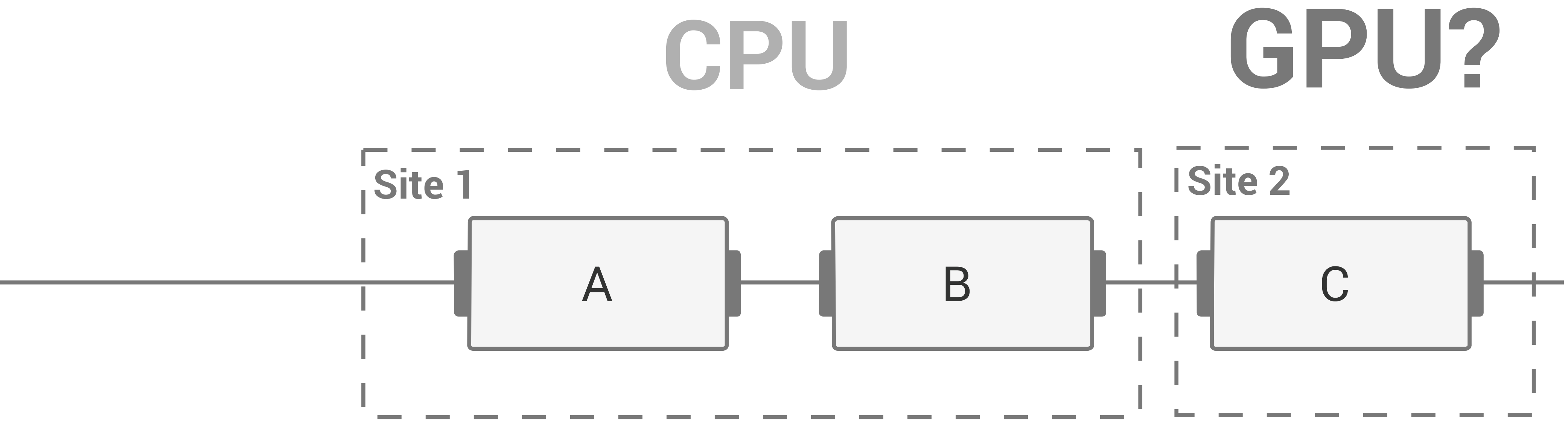
- **Parallelization of operators**



- **Distributed computation**



more computation resources



In DBMS?

Database Management Systems

GPUs in DMBS

- … Efficient co-processor
- … Might outperform CPUs for certain operations
- … Computations are highly parallel (SIMD)
- … Huge corpus on research results

Some conclusions

- Data **transfer costs** to and from graphic card **are critical**
- **Operation** should match GPU architecture (e.g., branch free)
- **Operation must be expensive** enough to amortize transfer costs
- **Column-oriented** architectures **save transfer costs**

GPU Acceleration for Data Stream Processing

Challenges

Limited memory on graphic cards

VS

(time-based) windows can be huge

event representation (tuple) does not match

the GPU architecture

GPU-ready Stream Processing

Our 1st contribution: Handle graphic card memory limitation for very large windows via *bucketing*

GPU-ready Stream Processing
Bucketing

We suggest

Portioning streams of variable-length window of tuples

into a stream of "*Buckets*"

Bucket: fixed-size window portions with column-oriented event representation

GPU-ready Stream Processing
Bucketing (2)

Bucketing Operator

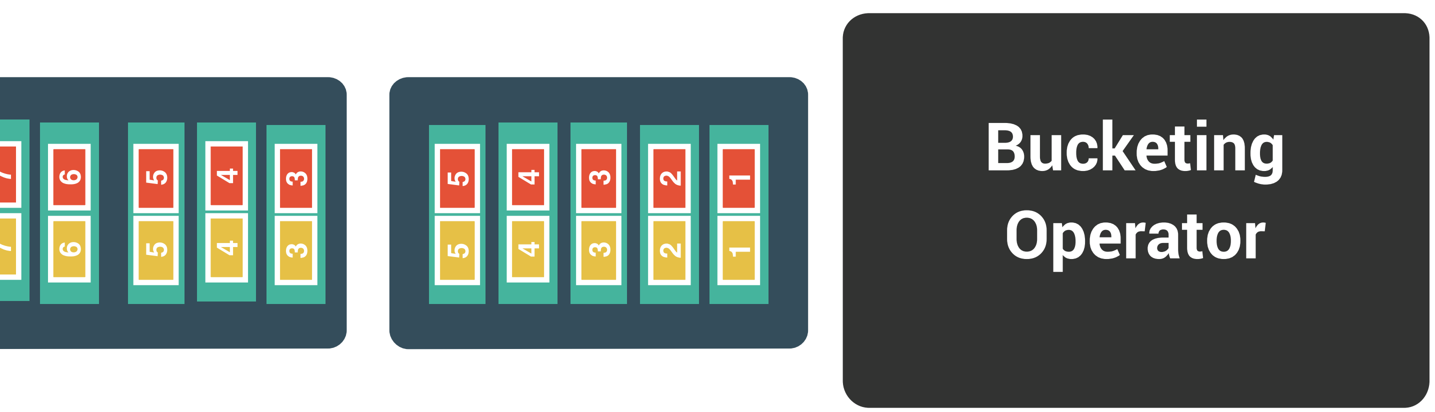
Bucket-at-a-Time
Let’s say bucket size 3

Let’s say bucket size 5

Bucket-at-a-Time

GPU-ready Stream Processing

Bucketing (2)



Bucket-at-a-Time

Let’s say bucket size 3

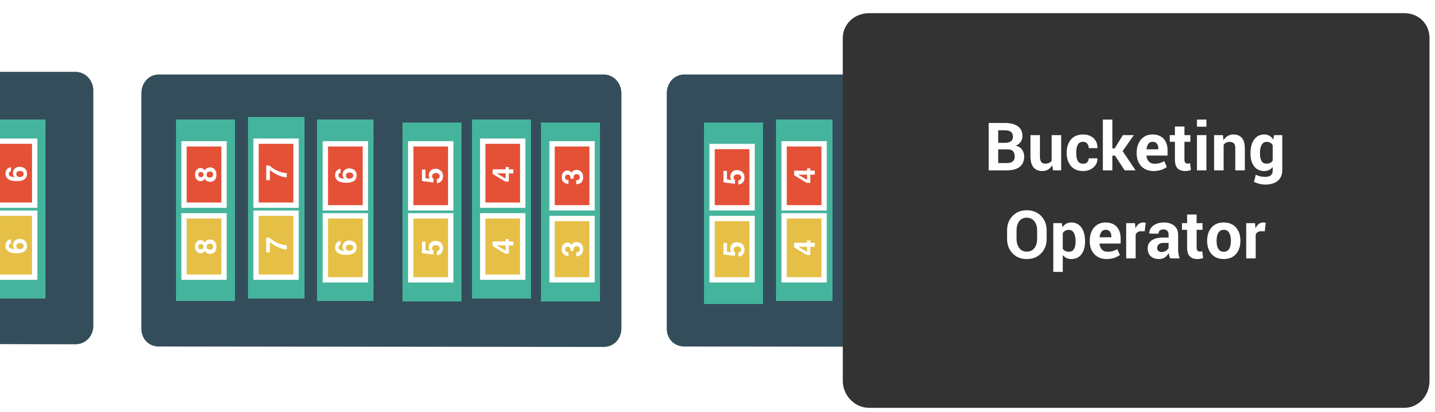
Let’s say bucket size 5

Bucket-at-a-Time

GPU-ready Stream Processing

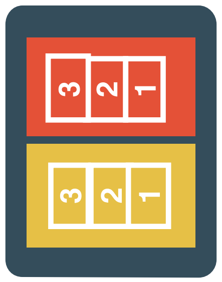
Bucketing (2)

3 events,
column-oriented



Bucket-at-a-Time

Let’s say bucket size 3

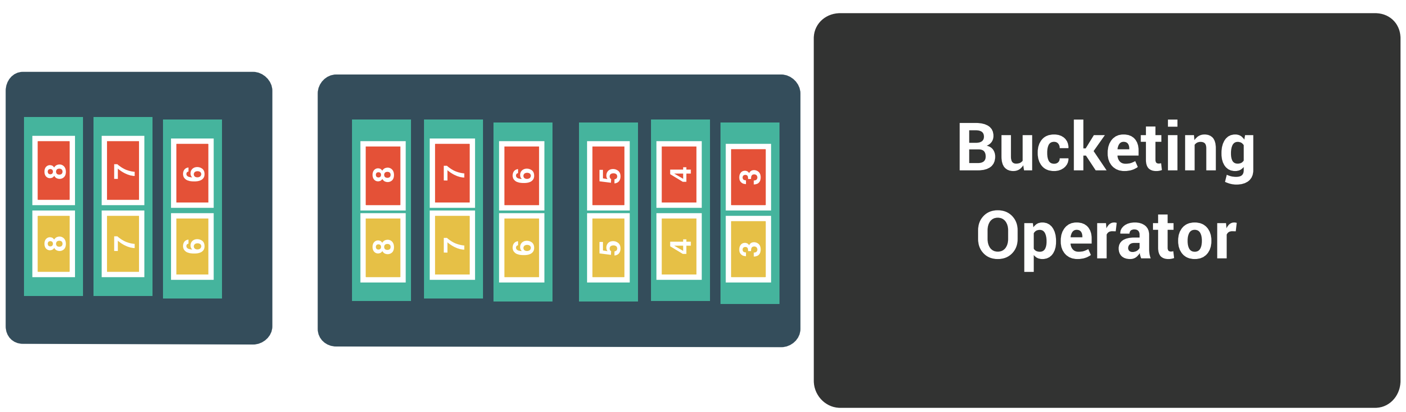


Let’s say bucket size 5

Bucket-at-a-Time

GPU-ready Stream Processing

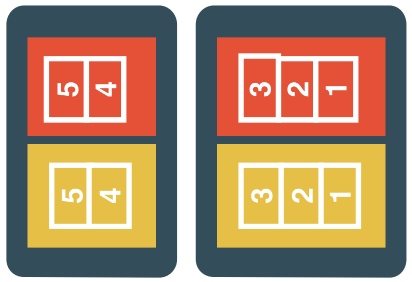
Bucketing (2)

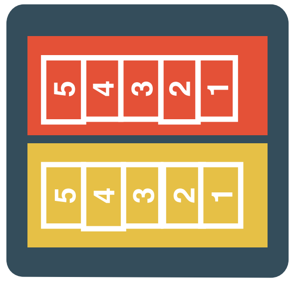


5 events,
column-oriented

Bucket-at-a-Time

Let’s say bucket size 3

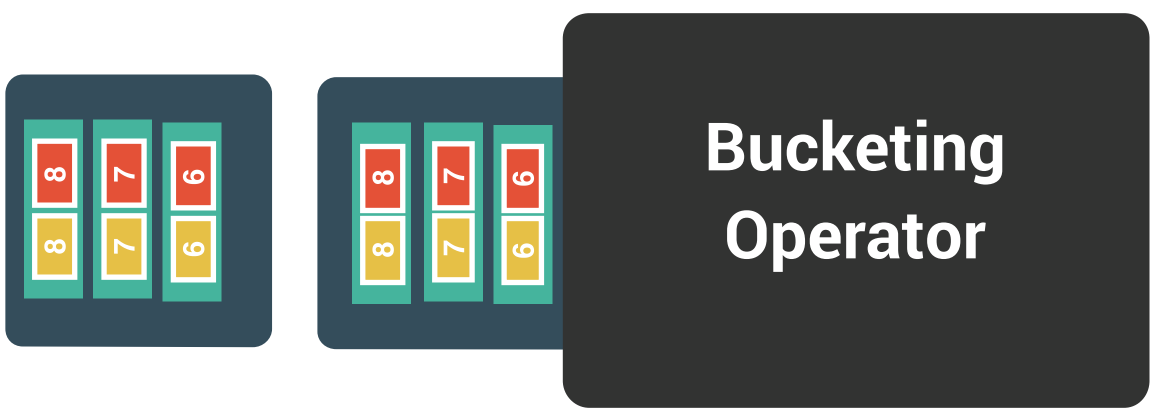




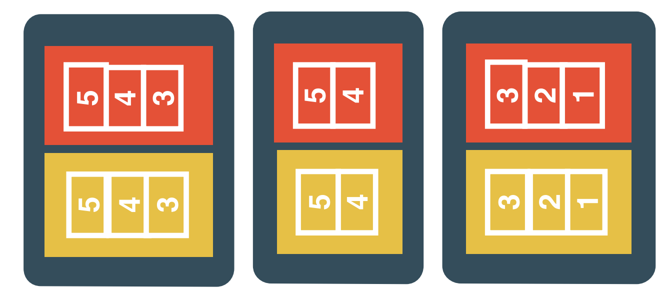
Let’s say bucket size 5

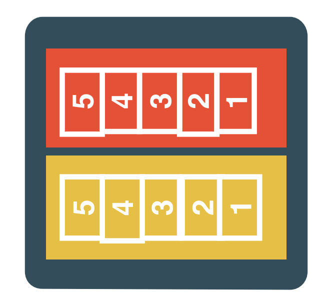
Bucket-at-a-Time

GPU-ready Stream Processing
Bucketing (2)



Bucket-at-a-Time
Let’s say bucket size 3

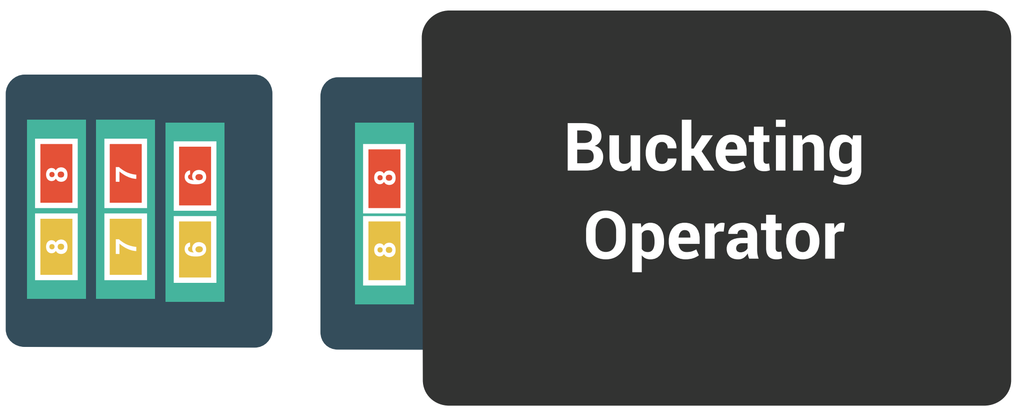




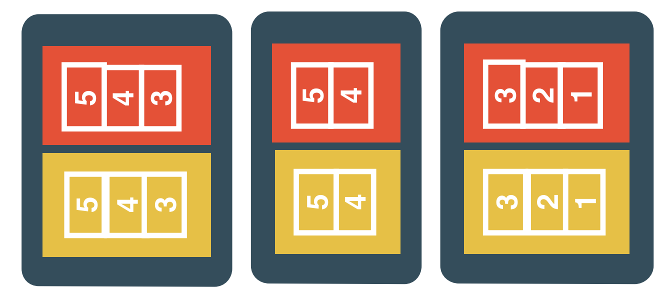
Let’s say bucket size 5

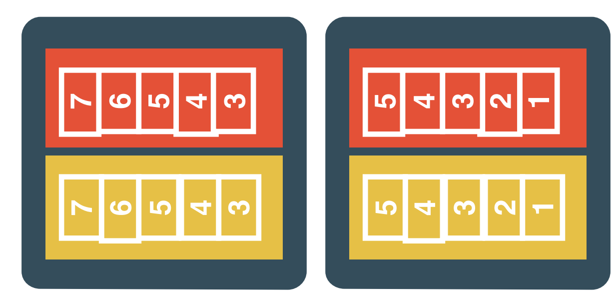
Bucket-at-a-Time

GPU-ready Stream Processing
Bucketing (2)



Bucket-at-a-Time
Let’s say bucket size 3



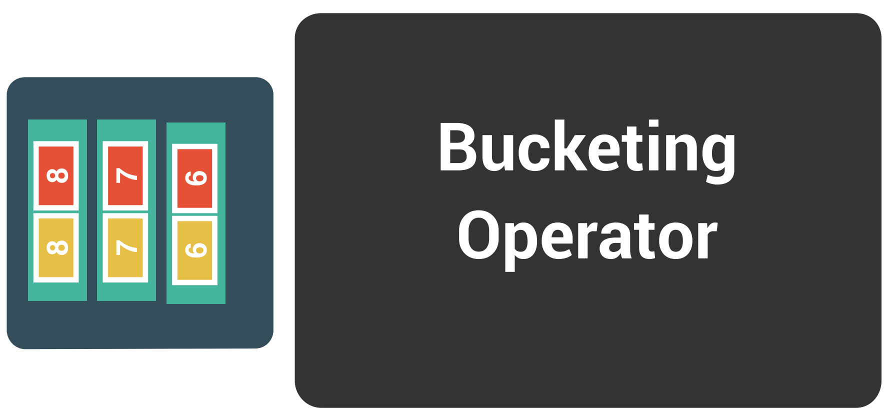


Let’s say bucket size 5

Bucket-at-a-Time

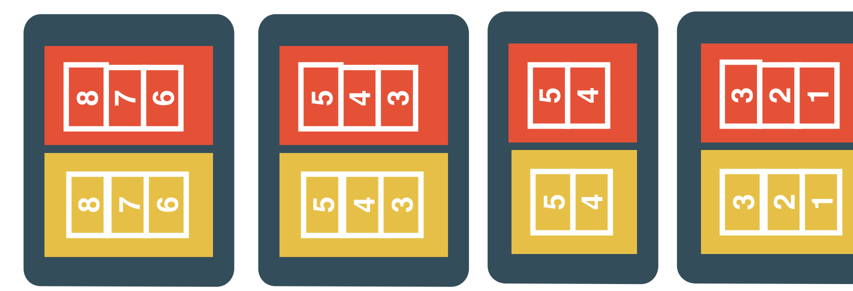
GPU-ready Stream Processing

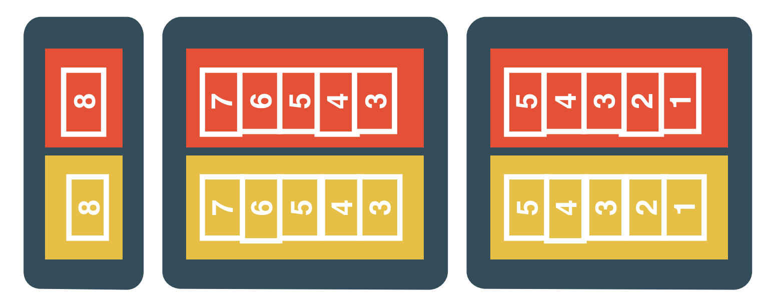
Bucketing (2)



Bucket-at-a-Time

Let's say bucket size 3



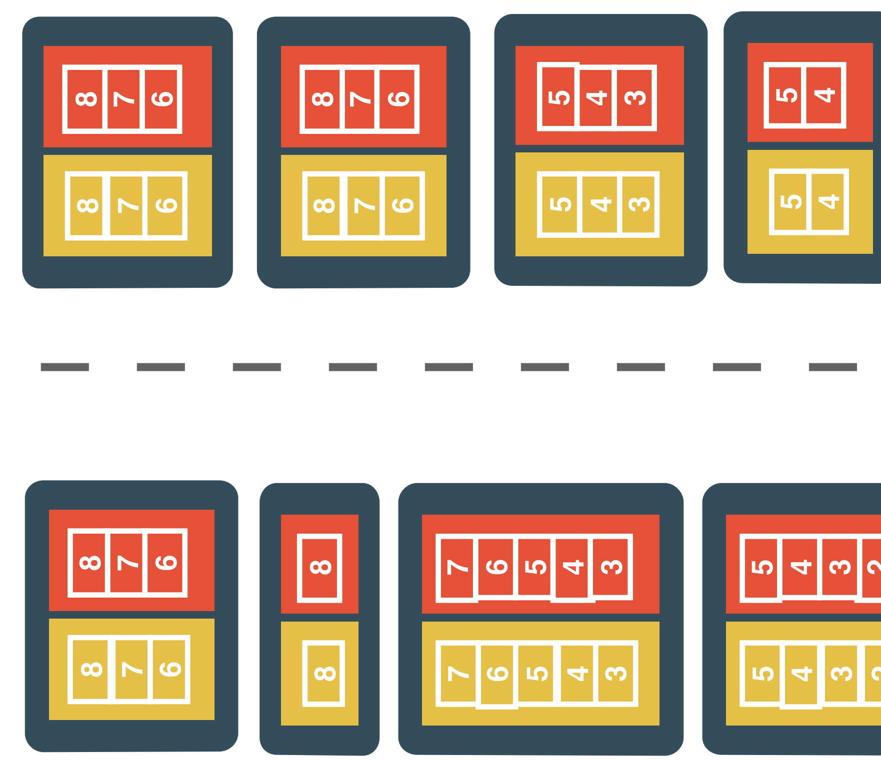


Let's say bucket size 5

Bucket-at-a-Time

GPU-ready Stream Processing
Bucketing (2)

Bucket-at-a-Time
Let’s say bucket size 3



Let’s say bucket size 5
Bucket-at-a-Time

GPU-ready Stream Processing
Benefits through Bucketing

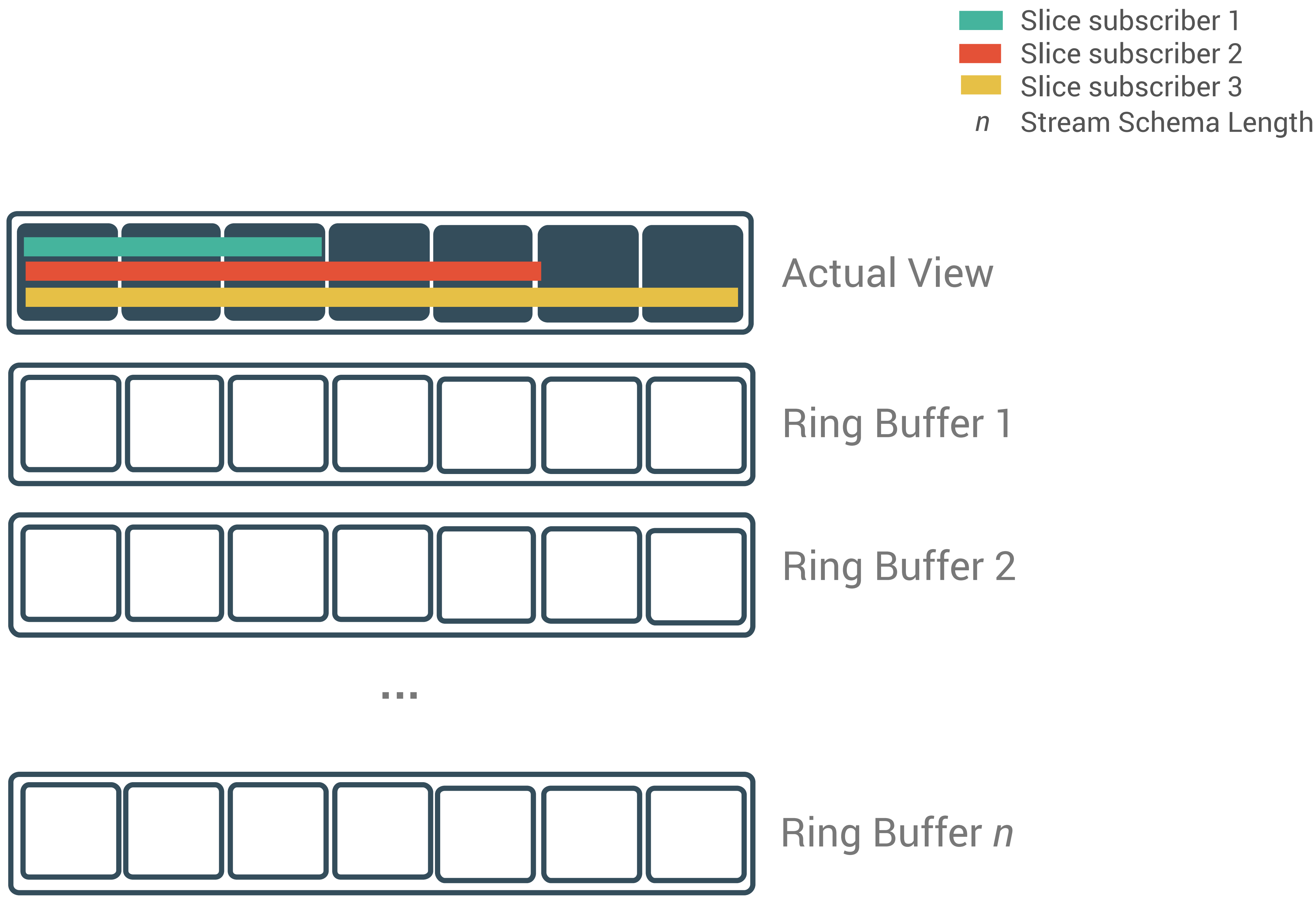
- **Each operator** requests its **own bucket size *k***
- The bucket size is **independent of** the actual **window length**
    - Memory allocation on graphic card has an upper bound for input
- Bucketing **flips event representation**
    - Processing entire columns
- **Window length > bucket size,** the **window is split** into portions
- **Single bucketing-operator** can be **subscribed by many** operators

GPU-ready Stream Processing
Buckets versus Windows

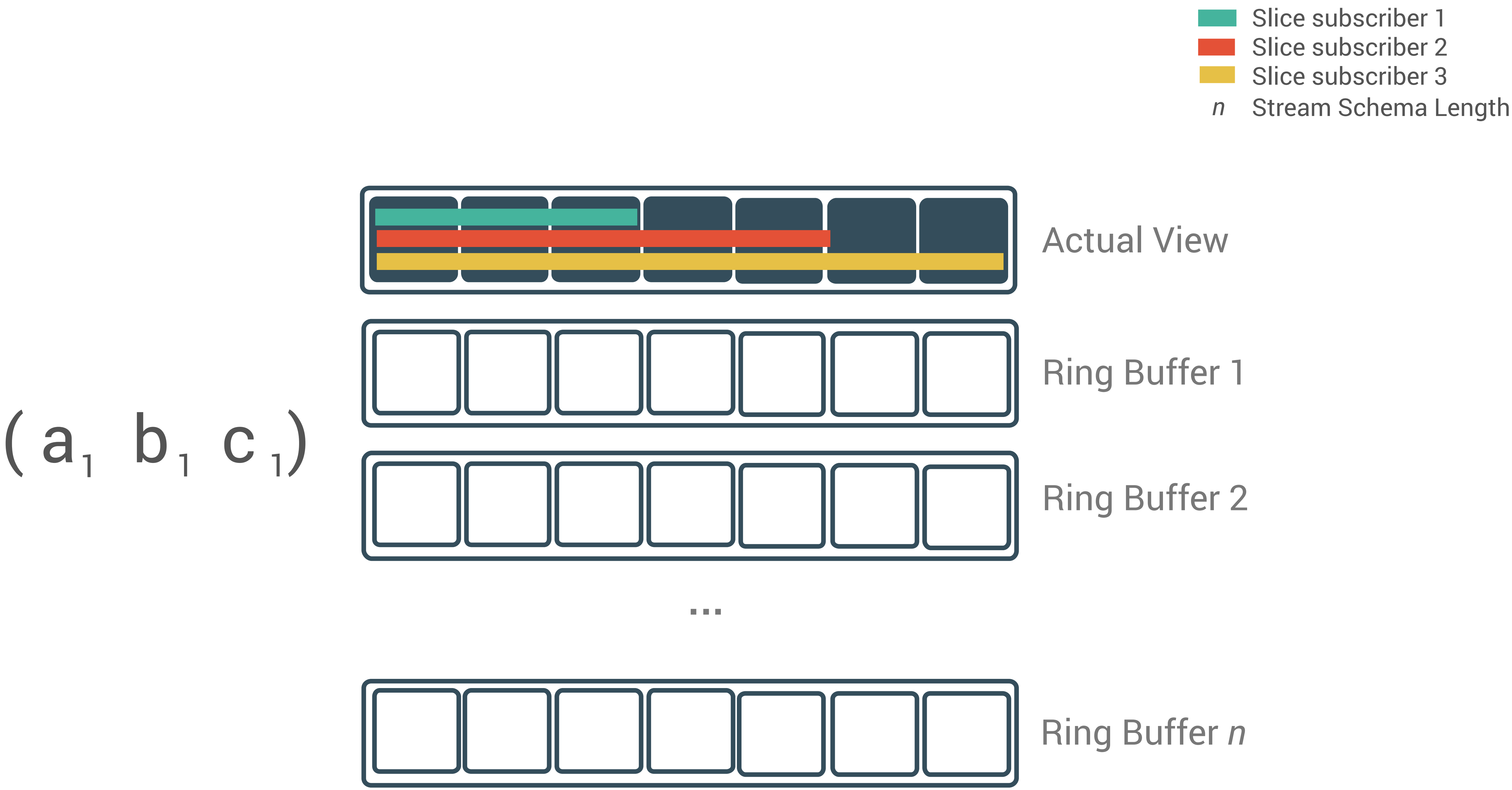
	Windowing	Bucketing
Purpose	▪ **Bounding** infinite **stream**	▪ **Portioning windows**
Consumes	▪ Stream of **events**	▪ Stream of **windows**
Produces	▪ Stream of **windows**	▪ Stream of **buckets**
#Events	▪ **Might be huge**	▪ **Has upper bound**
Events Represention	▪ **Tuples**	▪ **Column-wise**

GPU-ready Stream Processing

Achieve bucketing

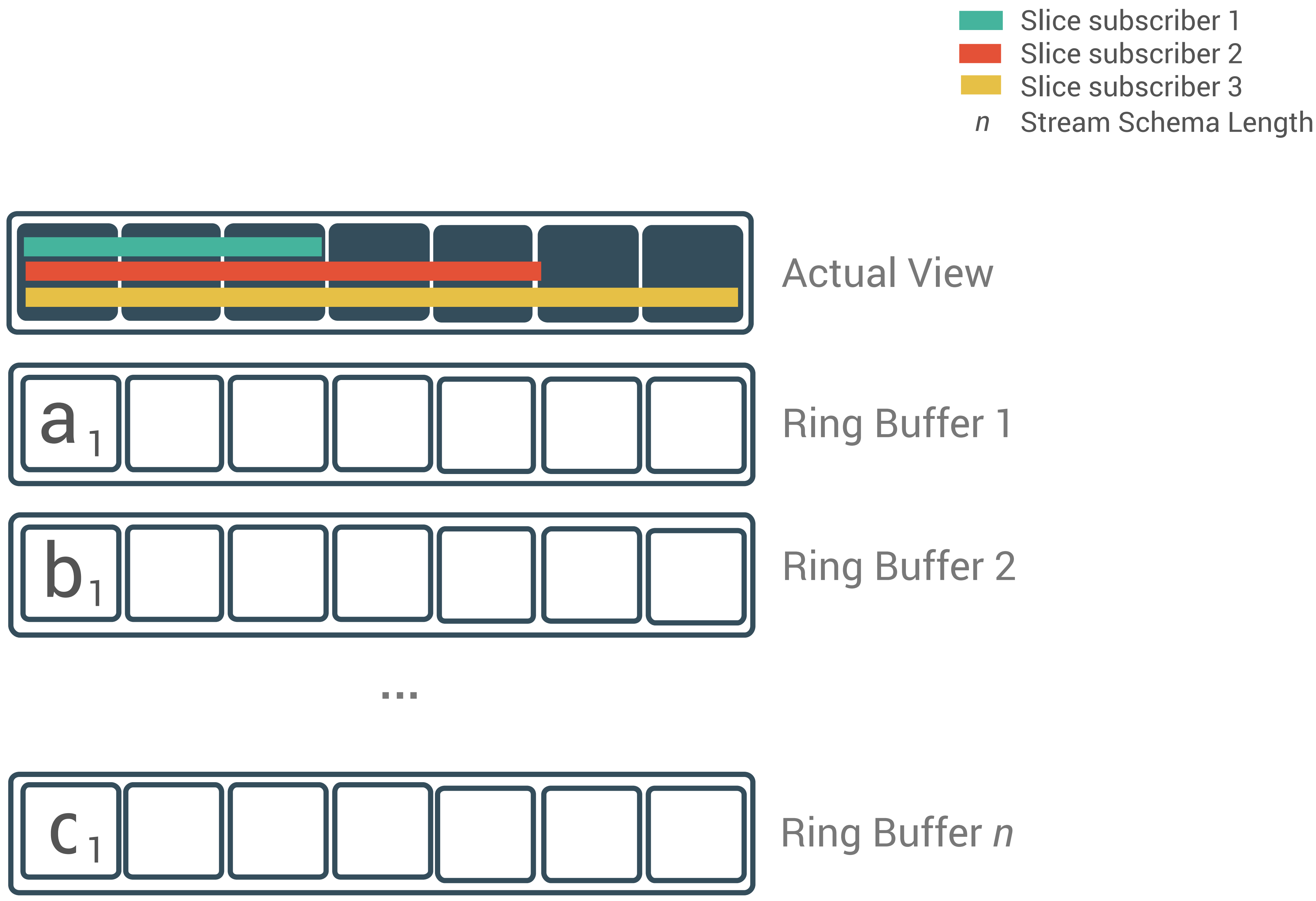


GPU-ready Stream Processing
Achieve bucketing



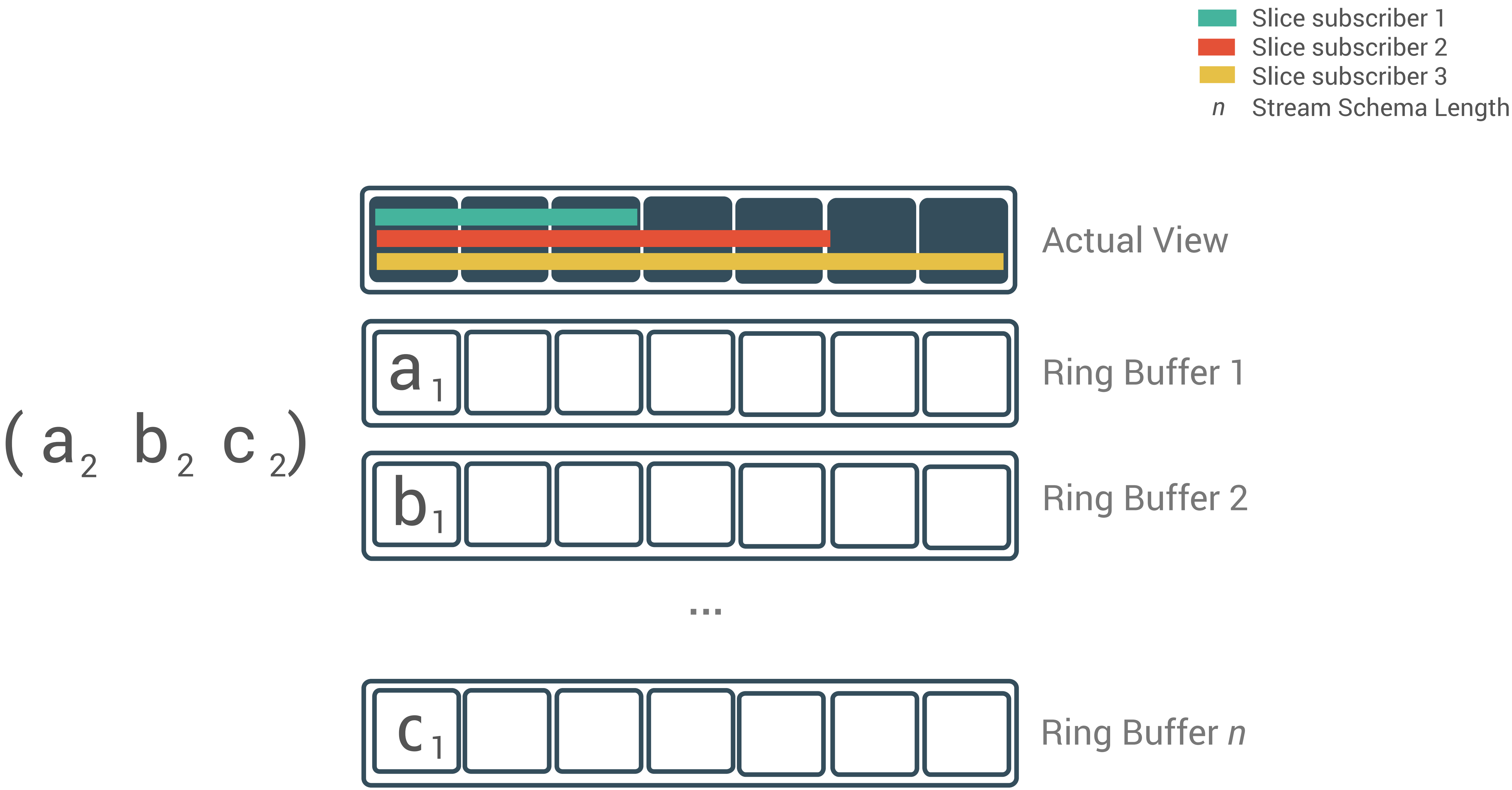
GPU-ready Stream Processing

Achieve bucketing

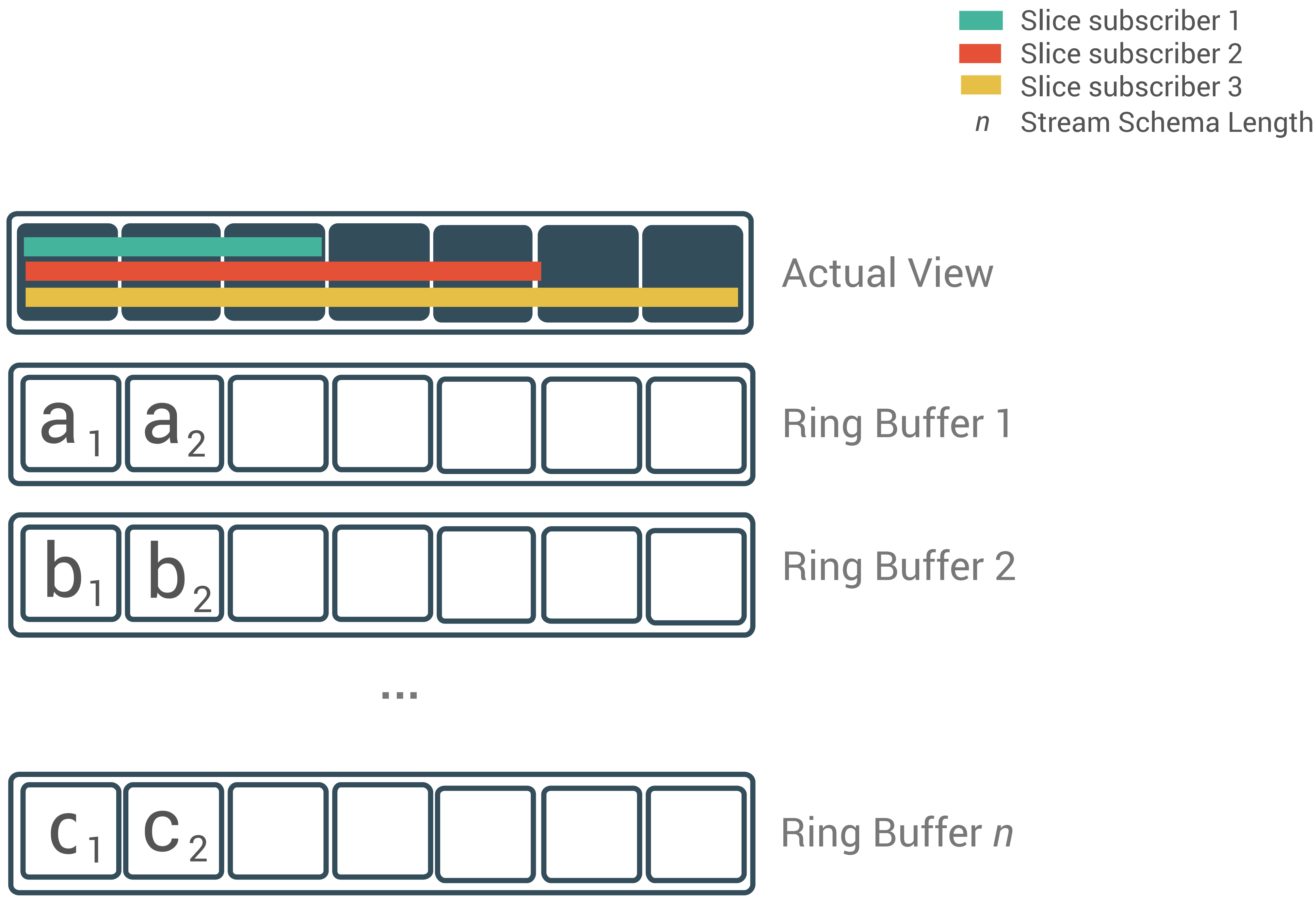


GPU-ready Stream Processing

Achieve bucketing

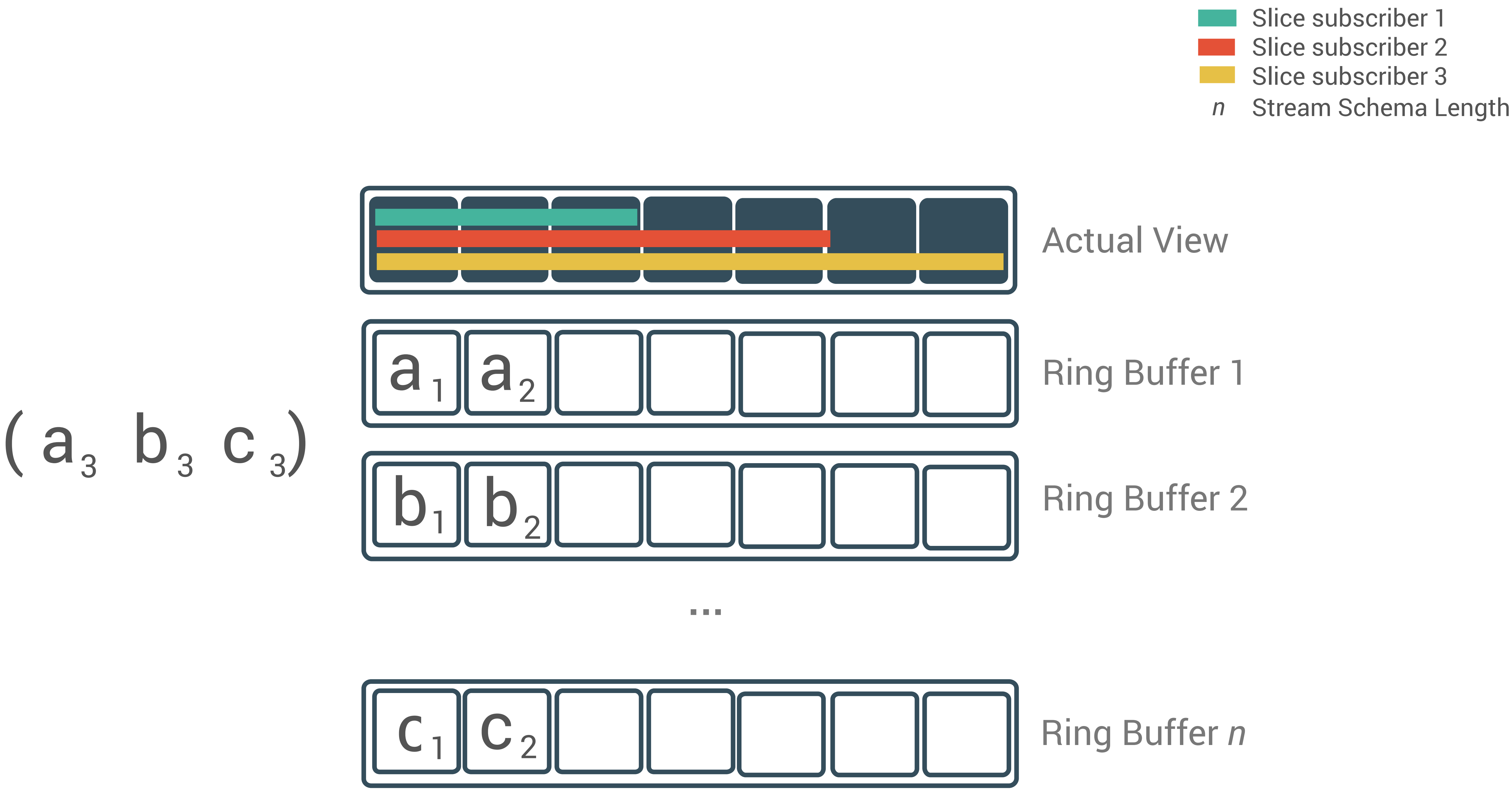


GPU-ready Stream Processing
Achieve bucketing



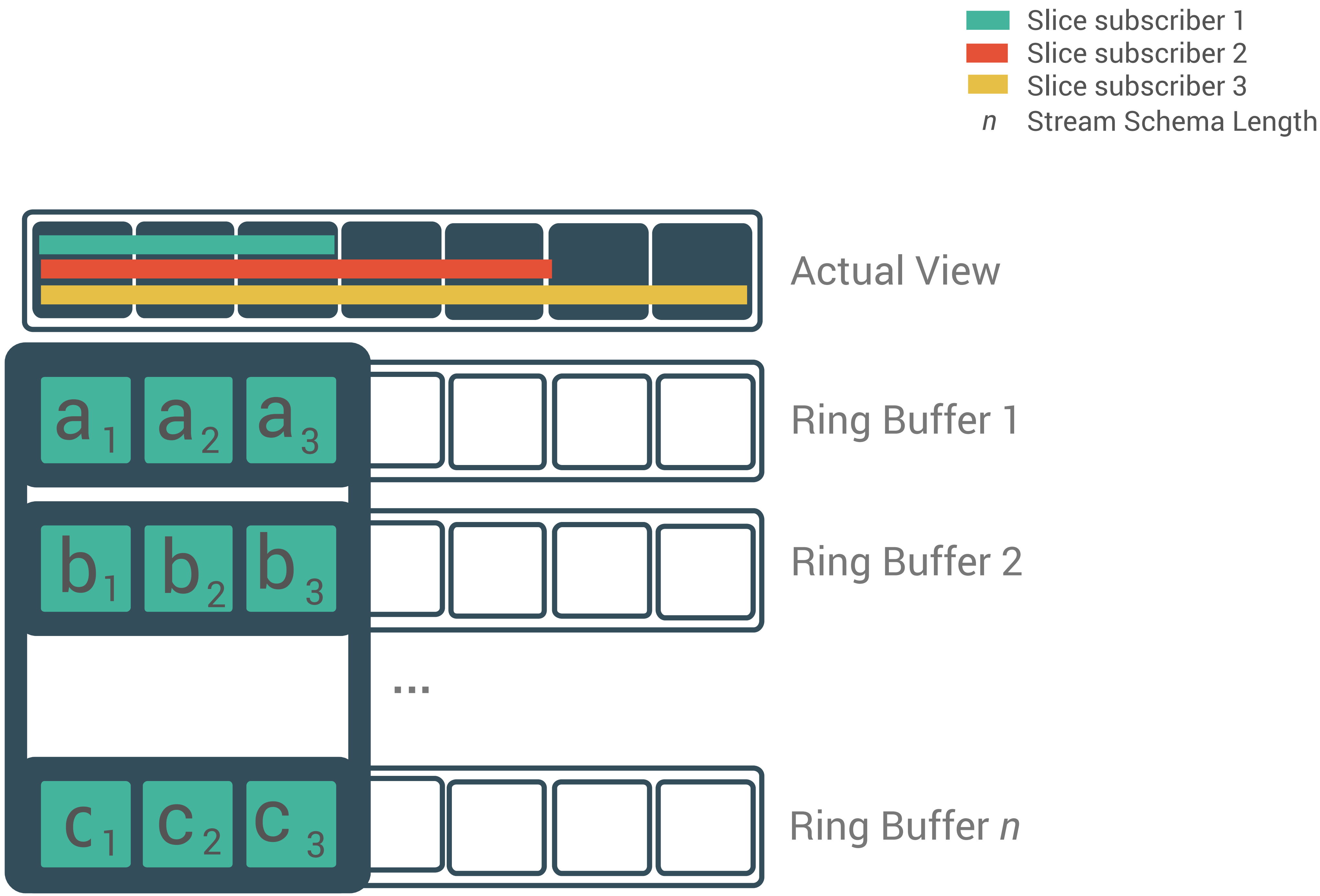
GPU-ready Stream Processing

Achieve bucketing

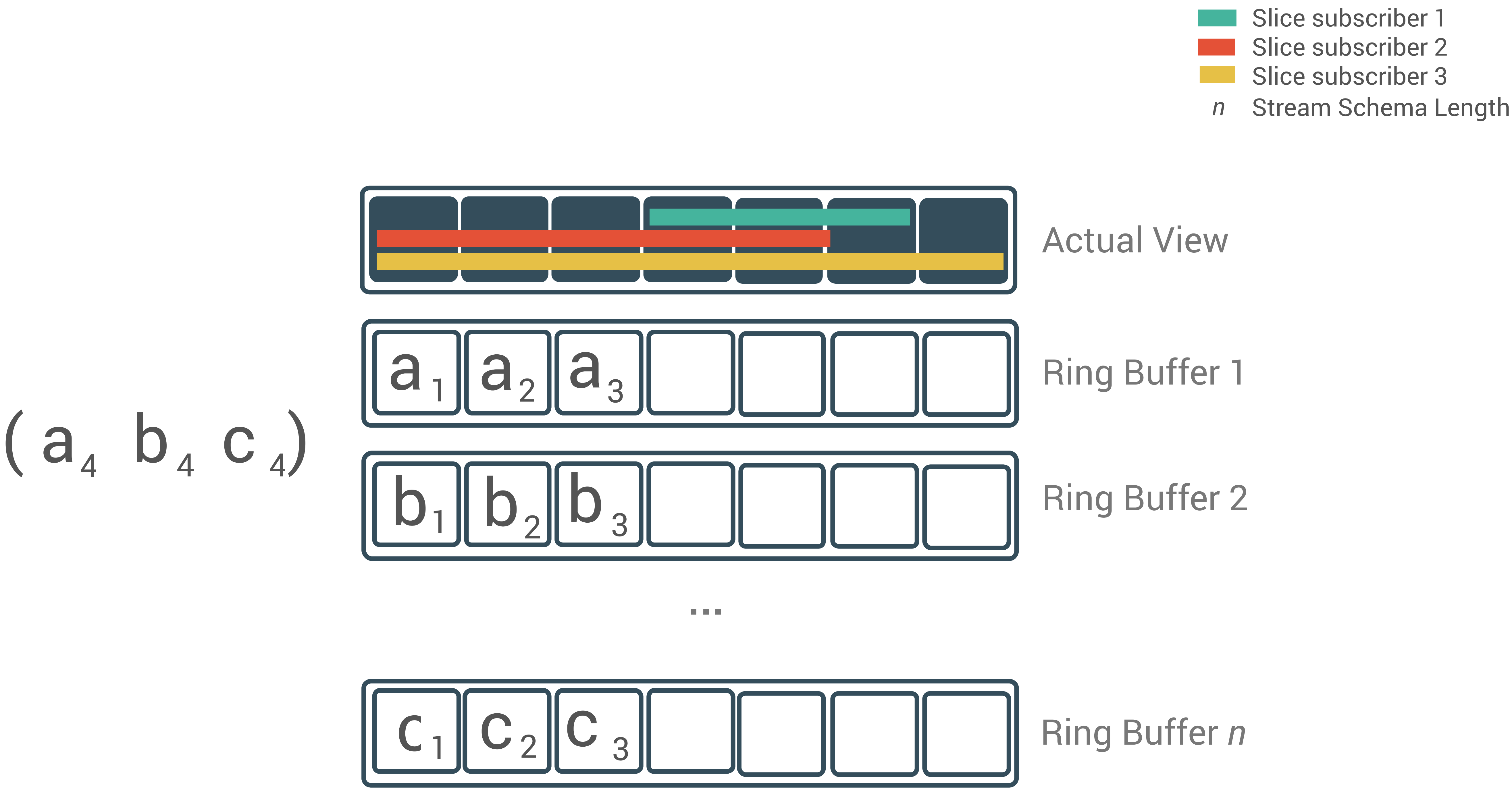


GPU-ready Stream Processing

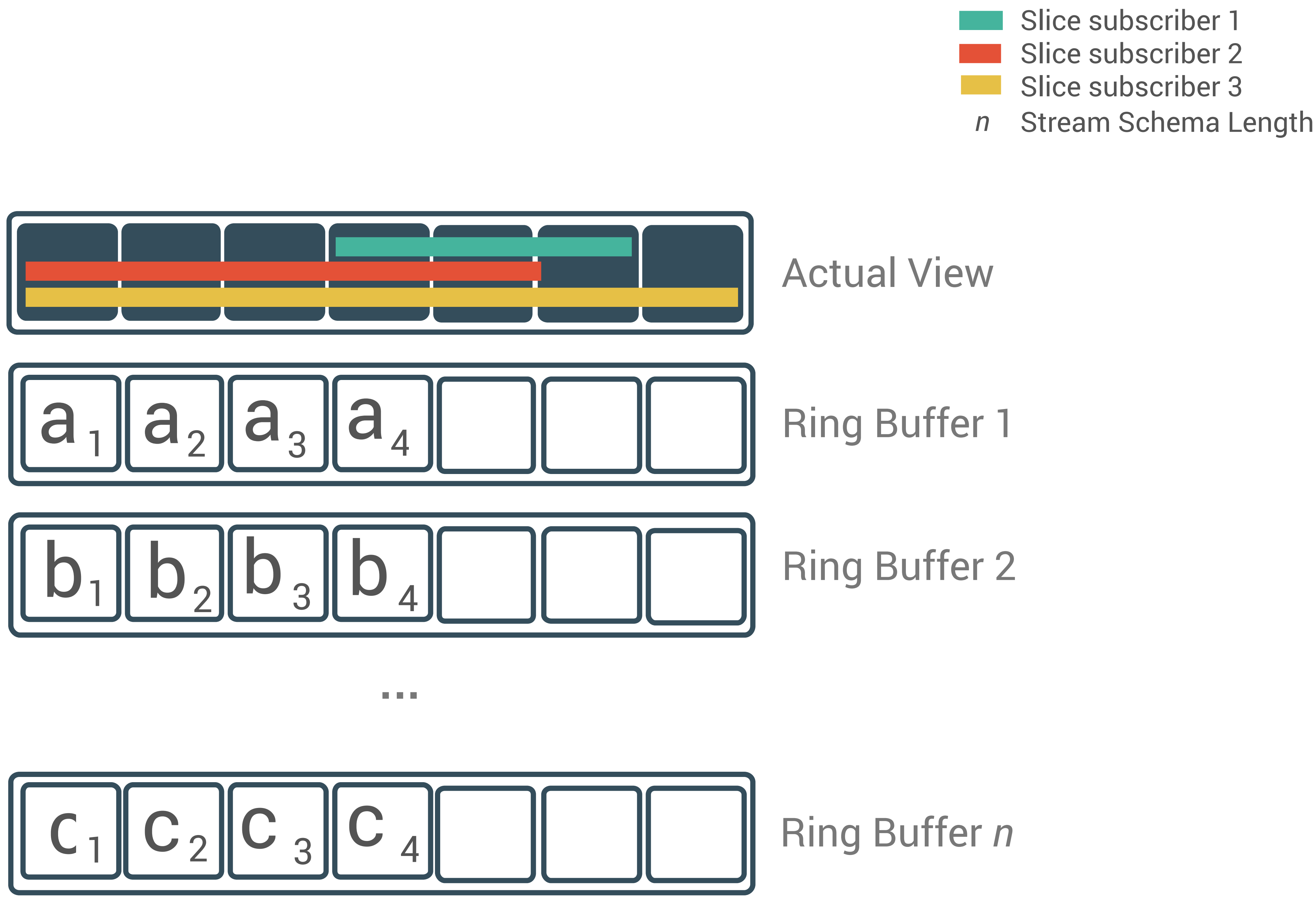
Achieve bucketing



GPU-ready Stream Processing
Achieve bucketing

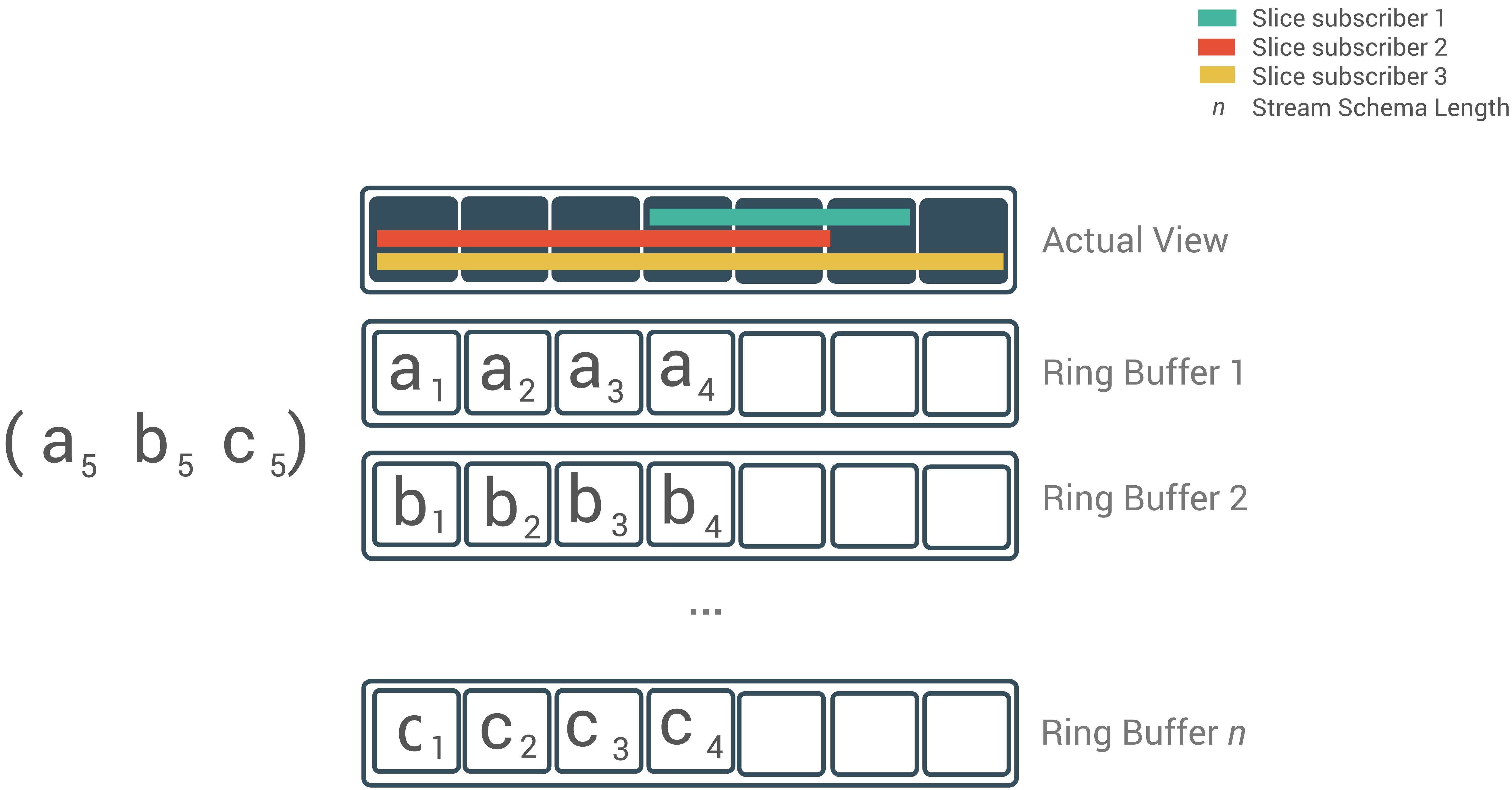


GPU-ready Stream Processing
Achieve bucketing

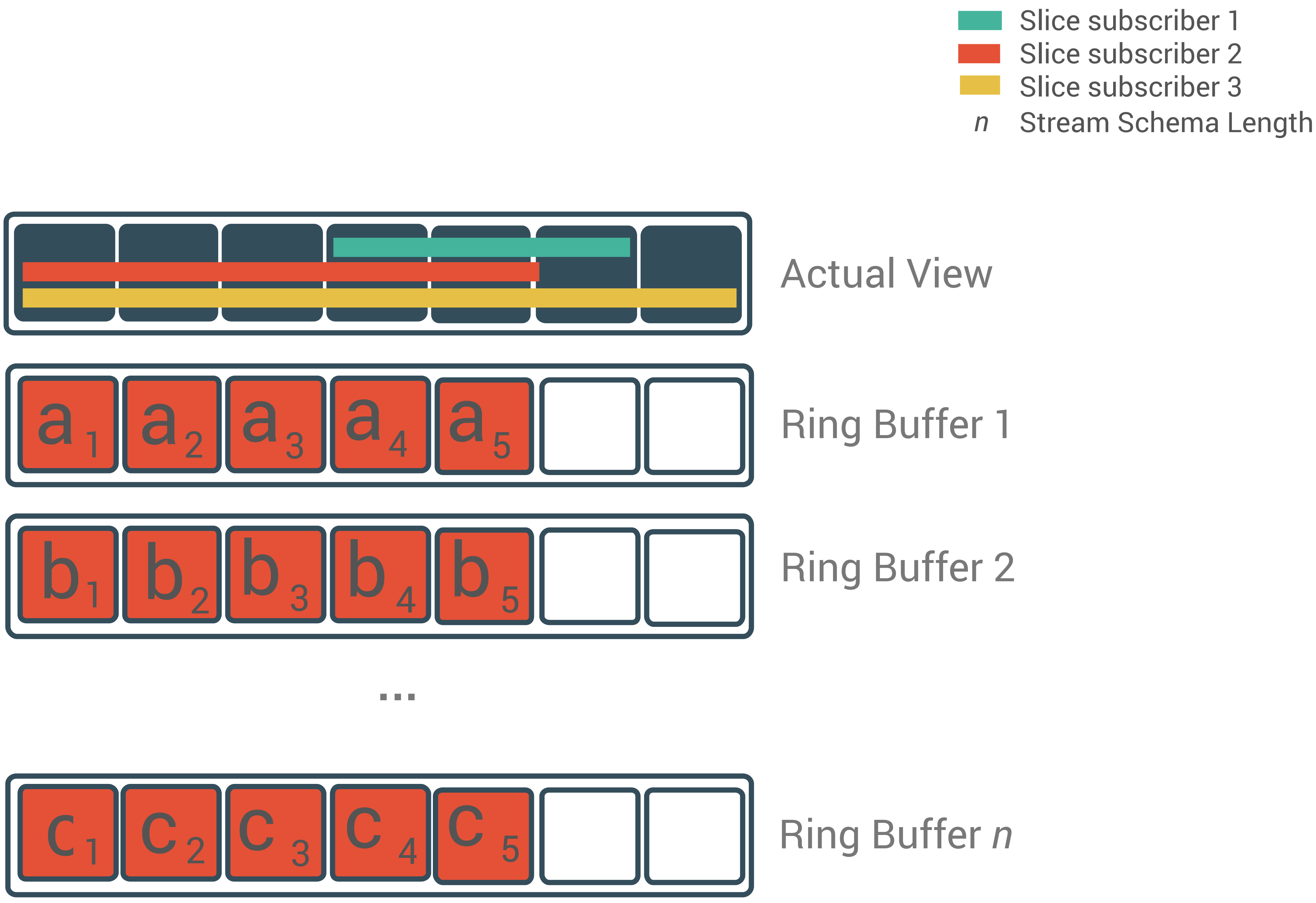


GPU-ready Stream Processing

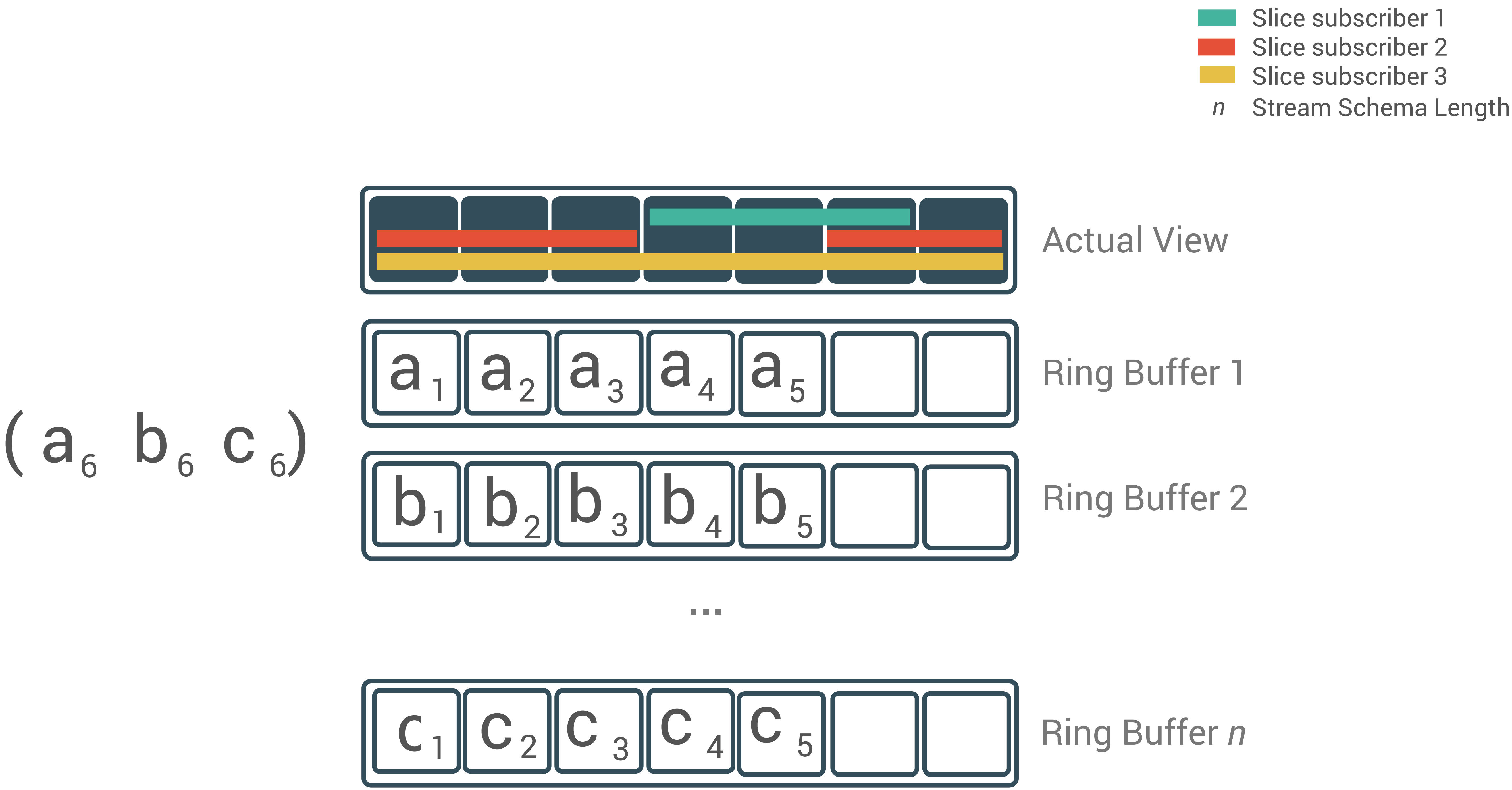
Achieve bucketing



GPU-ready Stream Processing
Achieve bucketing

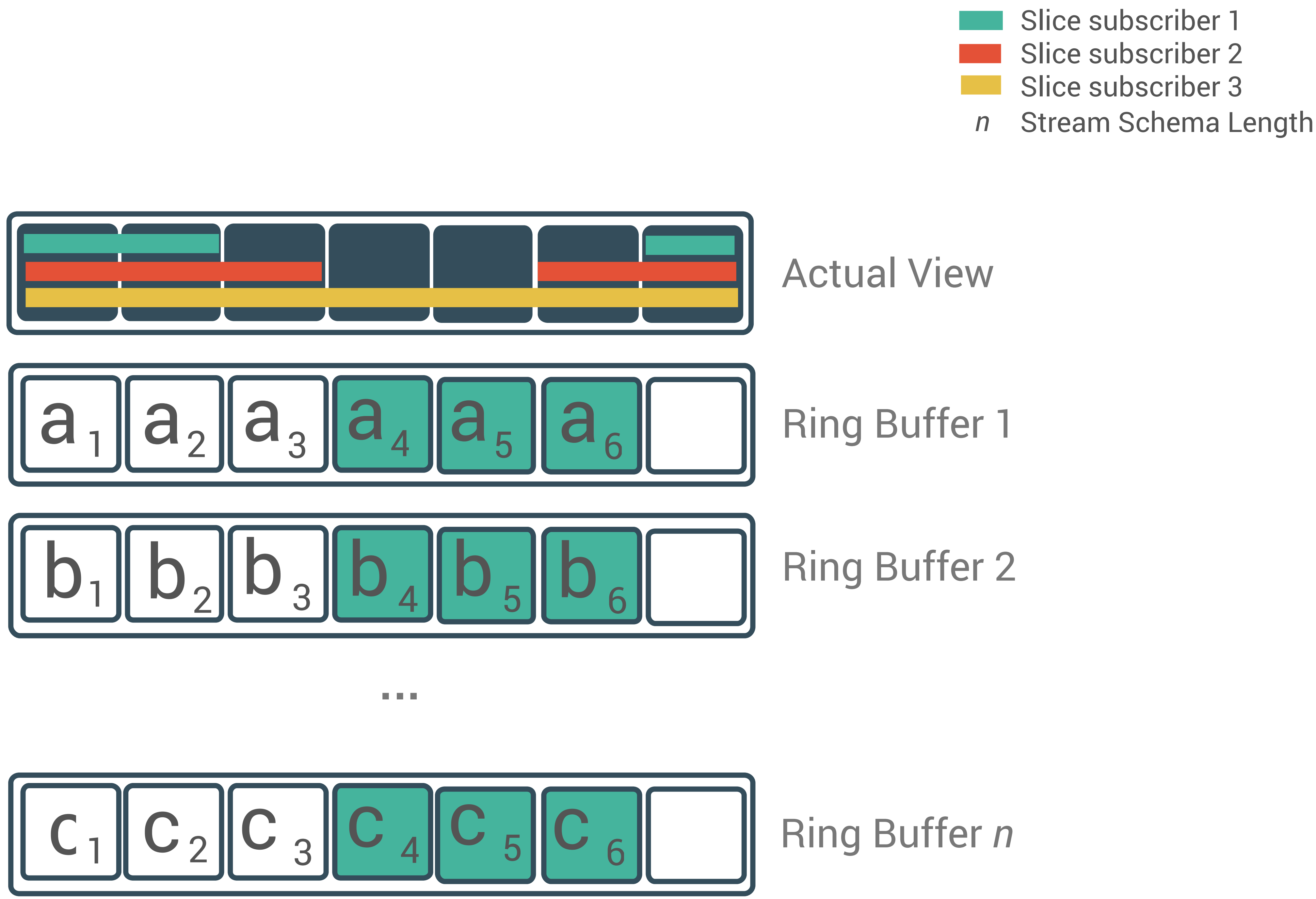


GPU-ready Stream Processing
Achieve bucketing



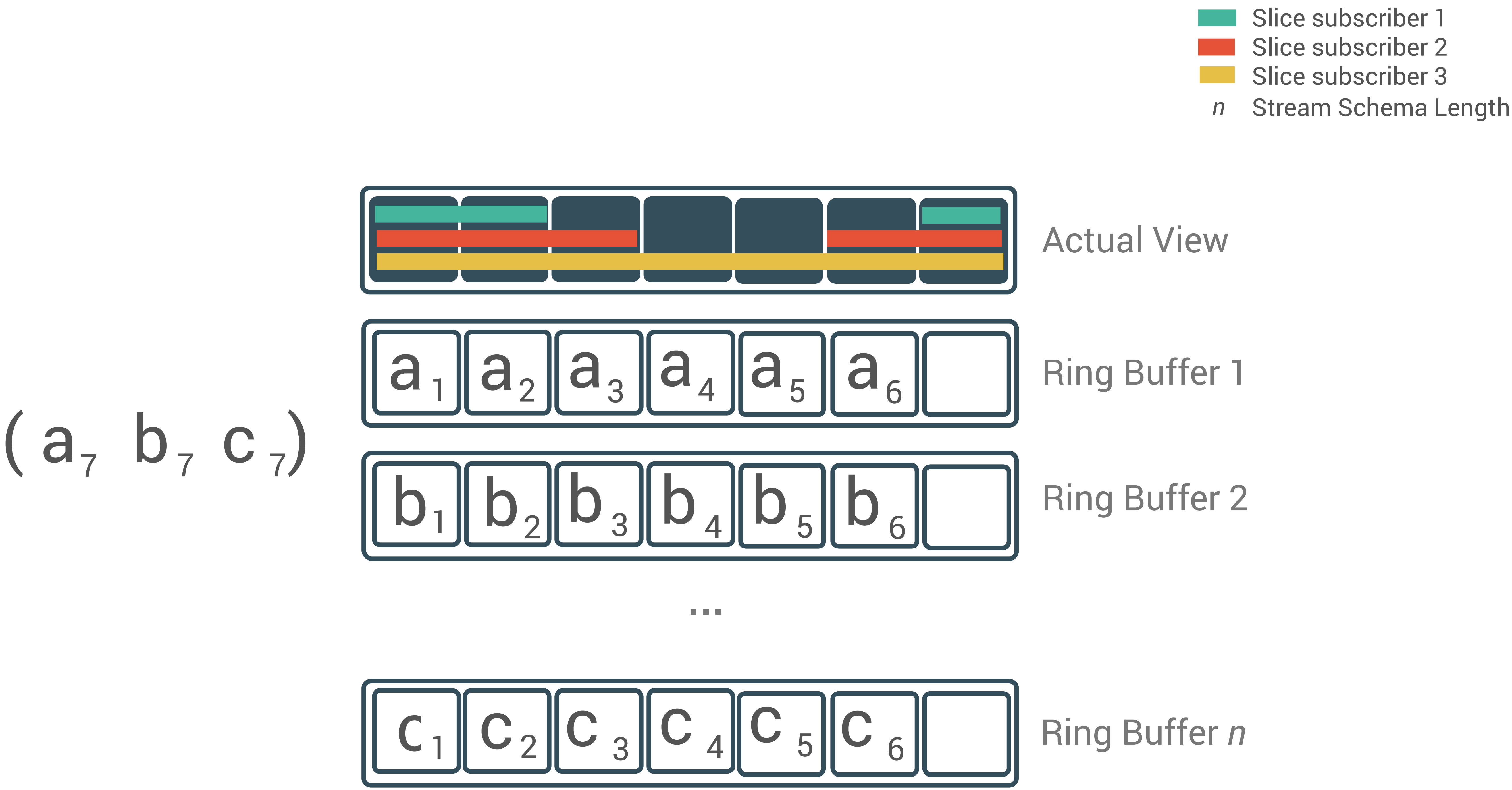
GPU-ready Stream Processing

Achieve bucketing



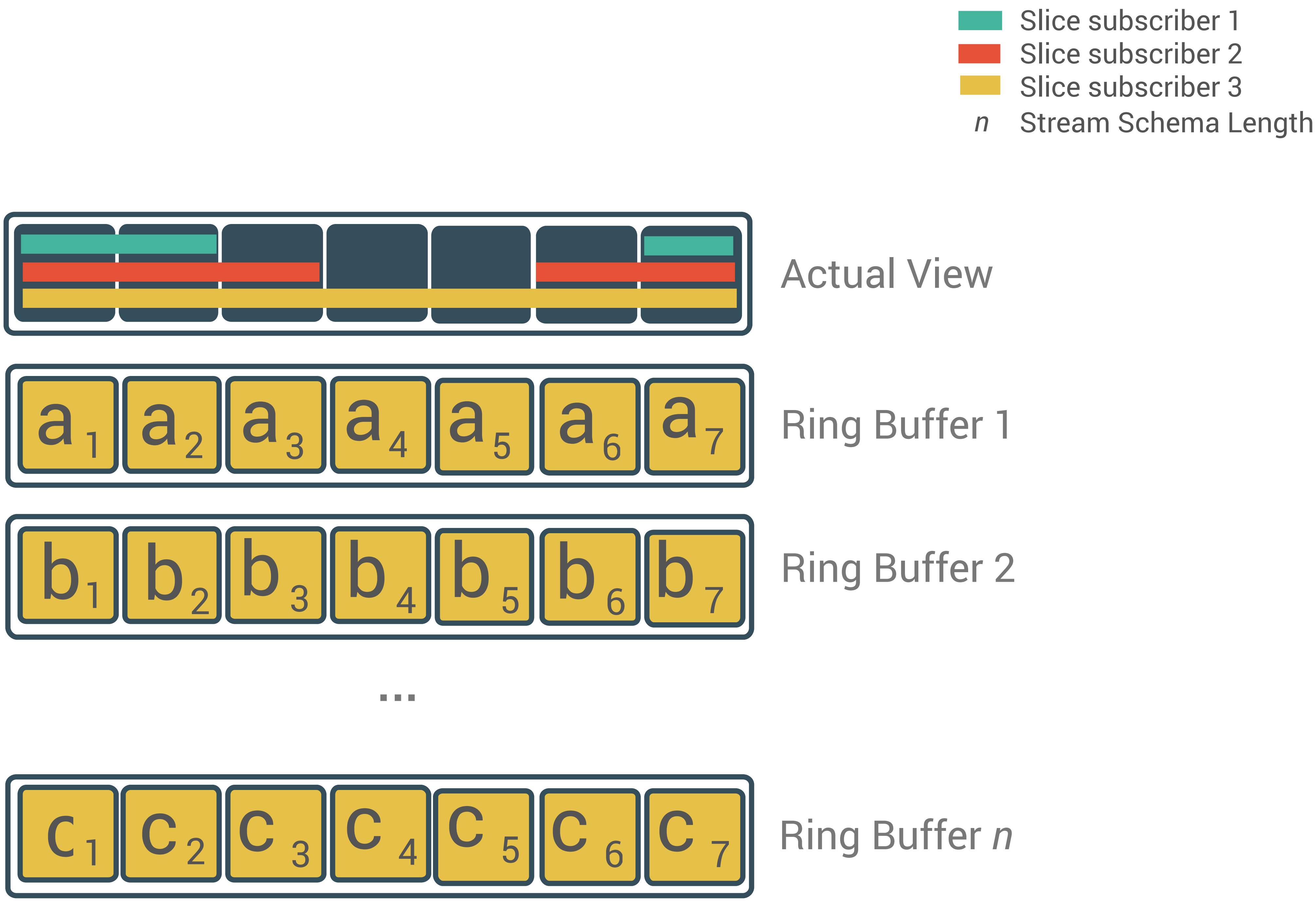
GPU-ready Stream Processing

Achieve bucketing



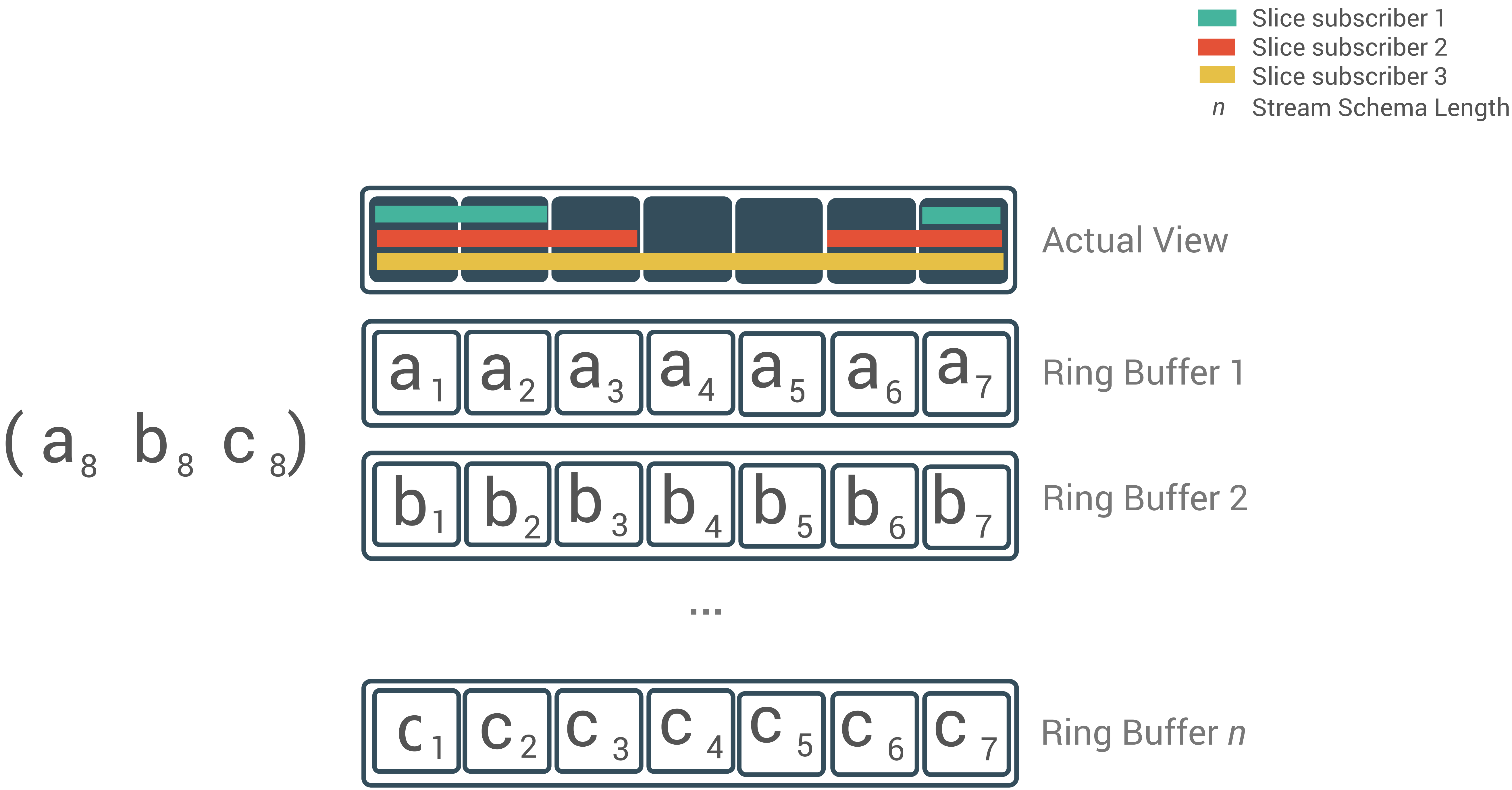
GPU-ready Stream Processing

Achieve bucketing

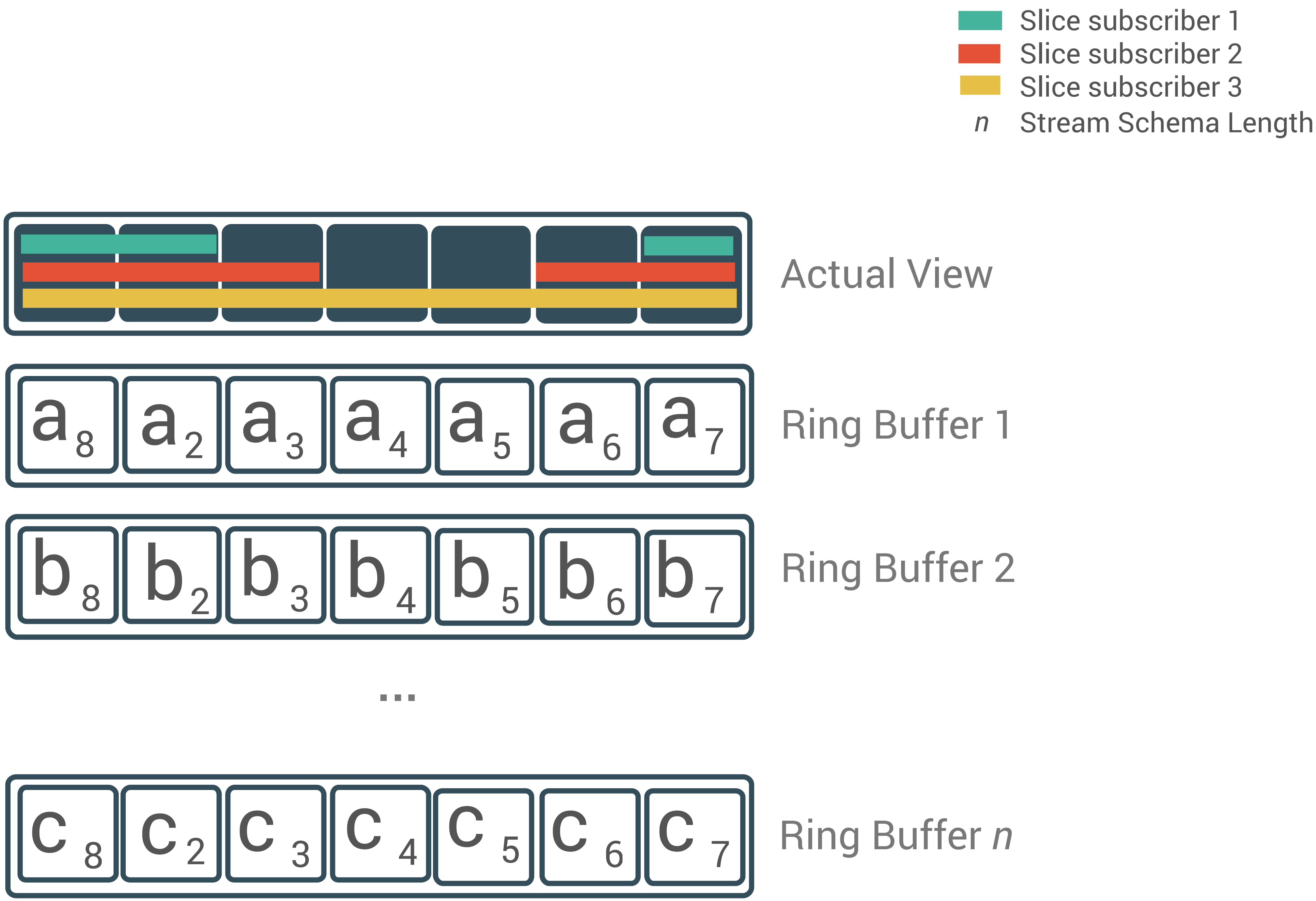


GPU-ready Stream Processing

Achieve bucketing

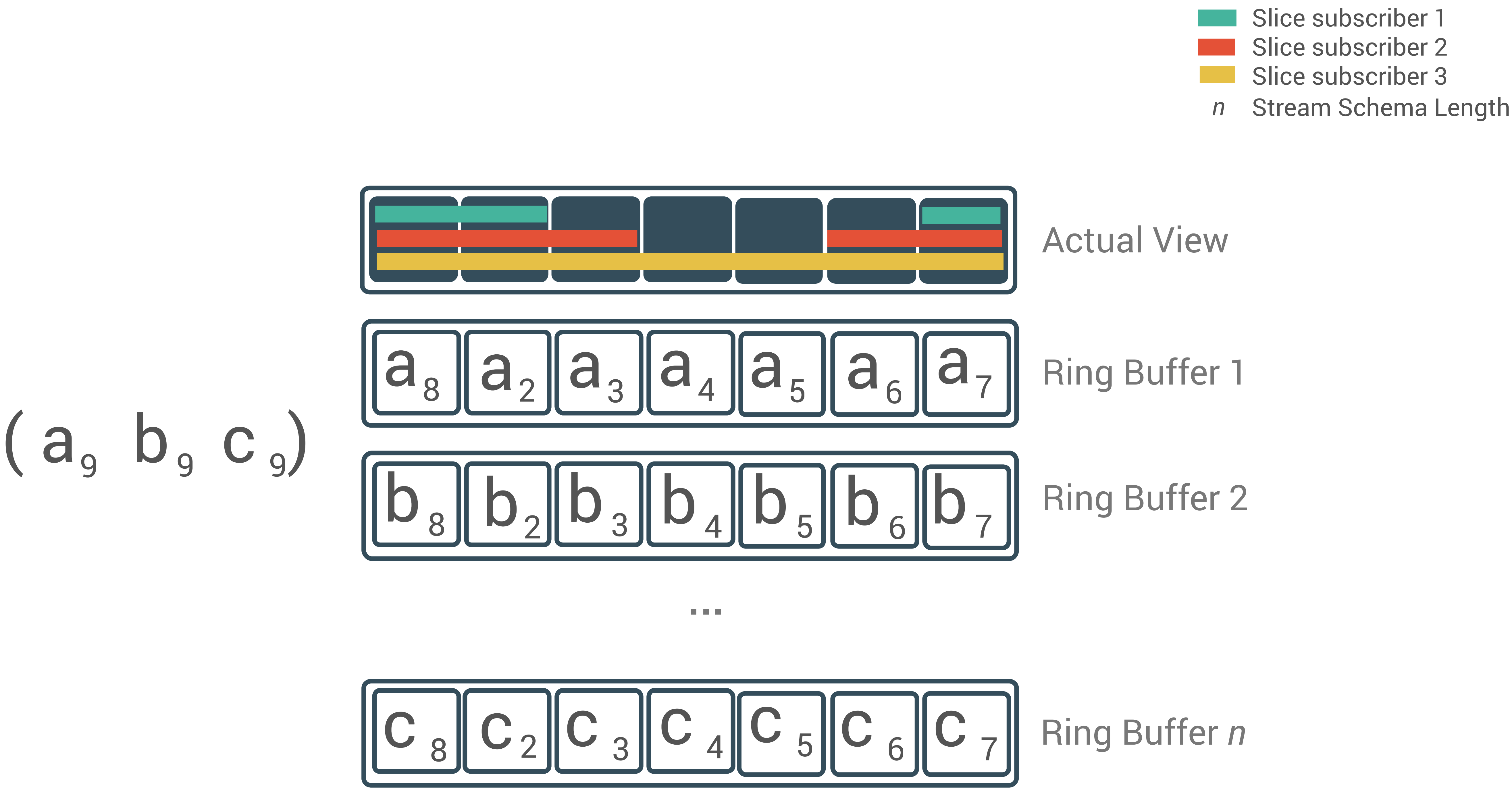


GPU-ready Stream Processing
Achieve bucketing



GPU-ready Stream Processing

Achieve bucketing



Open Research Challenges

Our 2st contribution: Identification of research challenges related to co-processing for Data Stream Processing

Open Research Challenges

Modern hardware and scheduling in Stream Processing

- **Other** specialized **co-processors** might be possible
  - Intel Xeon Phi or FPGAs for instance
- Optimized **algorithm** and executions **models for** the certain **co-processor**
- More than CPU-only Data Stream Processing:
  - Large physical **query execution plan space**
- **Find** best performance for a
  - **Logic plan** and
  - **Load sharing** between devices

***Further research** should be investigated to **find limitations** and **benefits** for applying modern hardware here*

Conclusion

Conclusion

Bucketing windows enables GPU-ready Data Stream Processing for very large windows

- Memory allocation has upper bound for input (fixed-size)
- Reduces transfer costs (column-selection)

We present an approach to achieve bucketing

- Separat operator, independent of SPS's tuple-at-a-time or batch-at-a-time support
- Ring buffer per attribute plus per-subscriber slice
- Enables processing of large-scale windows on limited graphic card memory
  - No fallback to CPU required

We identify research challenges for further co-processing in this context.

- Other co-processors with specialized algorithm — limitations and benefits
- Large search space for query plans (logical operator — devices — concrete algorithm) — optimizer