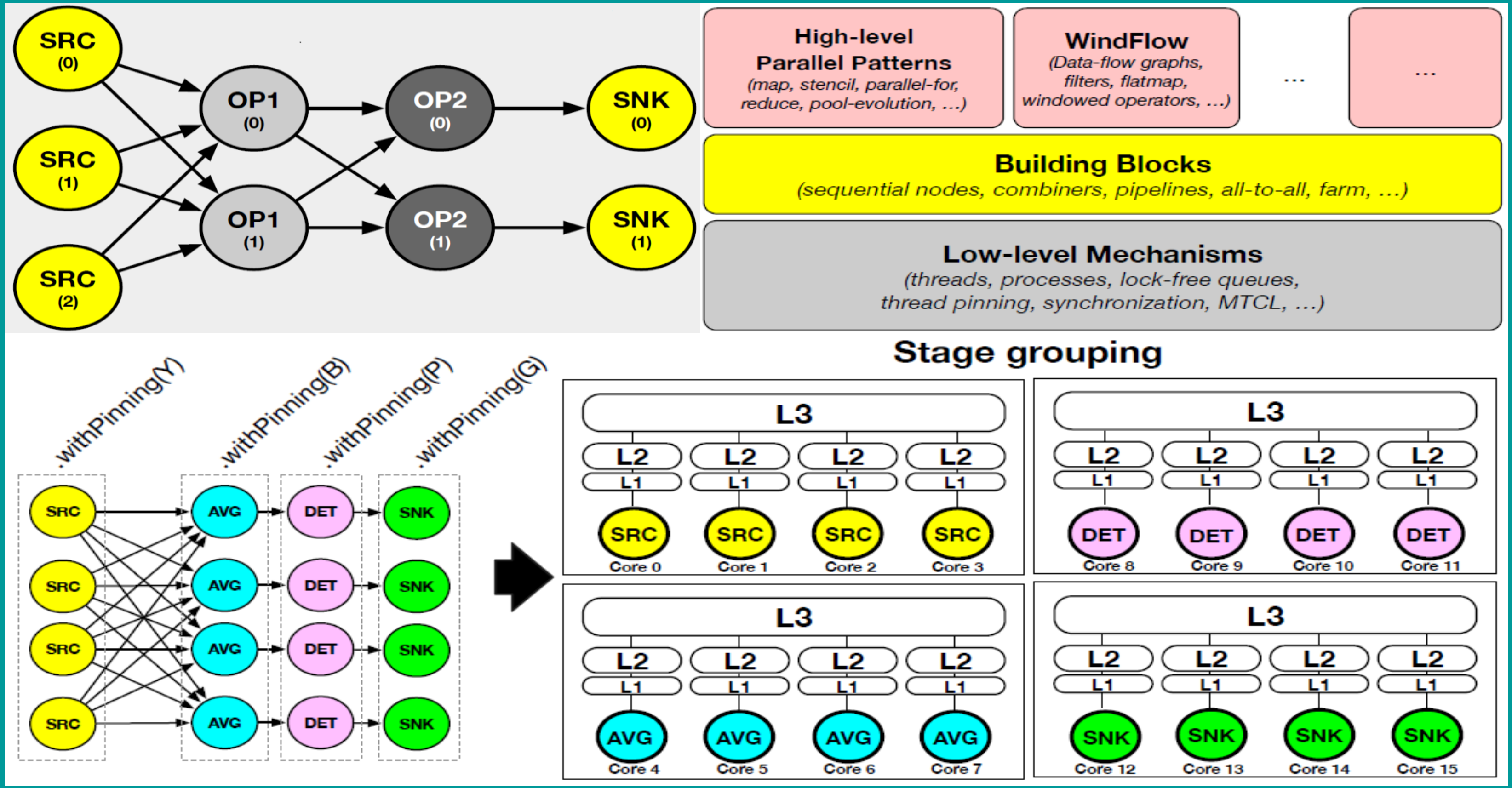



# Enabling Pinning Strategies for Stream Processing Applications on Multicores

Lorenzo Bindi, Salvatore D. D'Amico, Gabriele Mencagli, **Massimo Torquati**

Computer Science Department

Università di Pisa

HLPP 2025
18th International Symposium on High-level Parallel Programming and Applications

03/07/2025 Innsbruck, Austria

# Outline

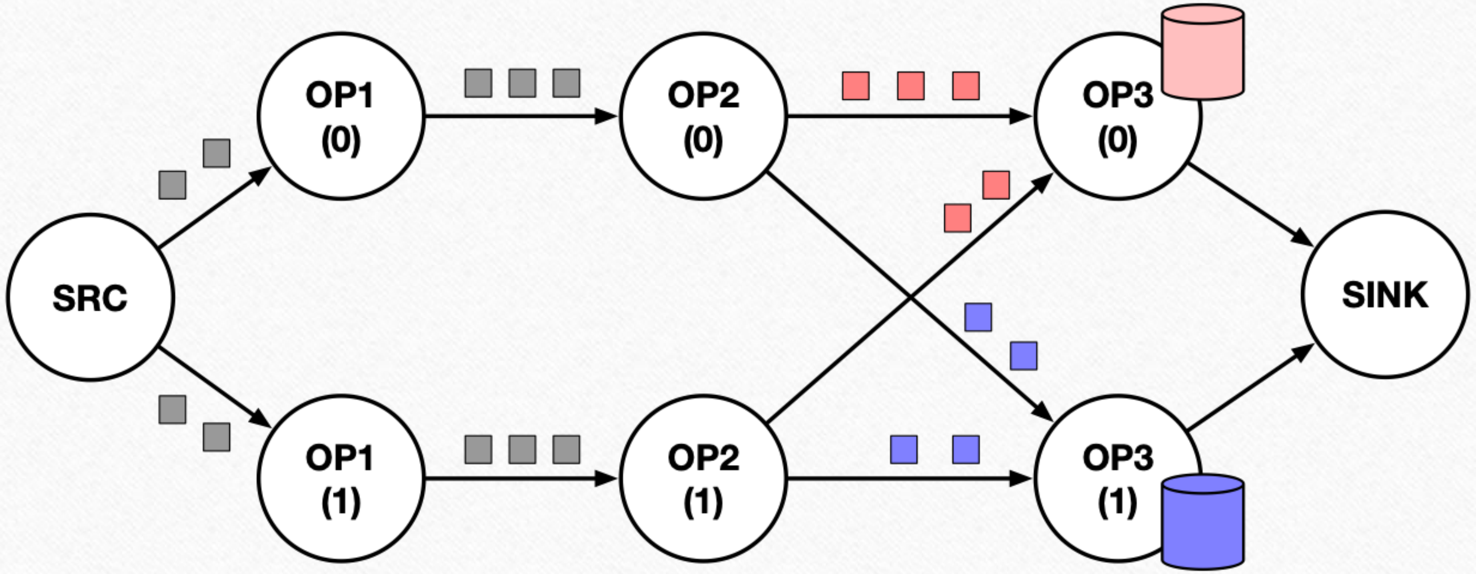
- Context
- Motivations of the work
- Background knowledge
  - FastFlow and WindFlow libraries
  - FastFlow thread-to-core pinning
- Work contributions
- Evaluation: experimental results
- Conclusions and Future Work

FastFlow

WINDFLOW

# Data Stream Processing (DSP)

- A computing paradigm that enables continuous data-stream processing to produce analytics, insights, and knowledge, continuously and in near real-time
- Applications (queries) expressed with data-flow graphs of **stateless** and **stateful** operators

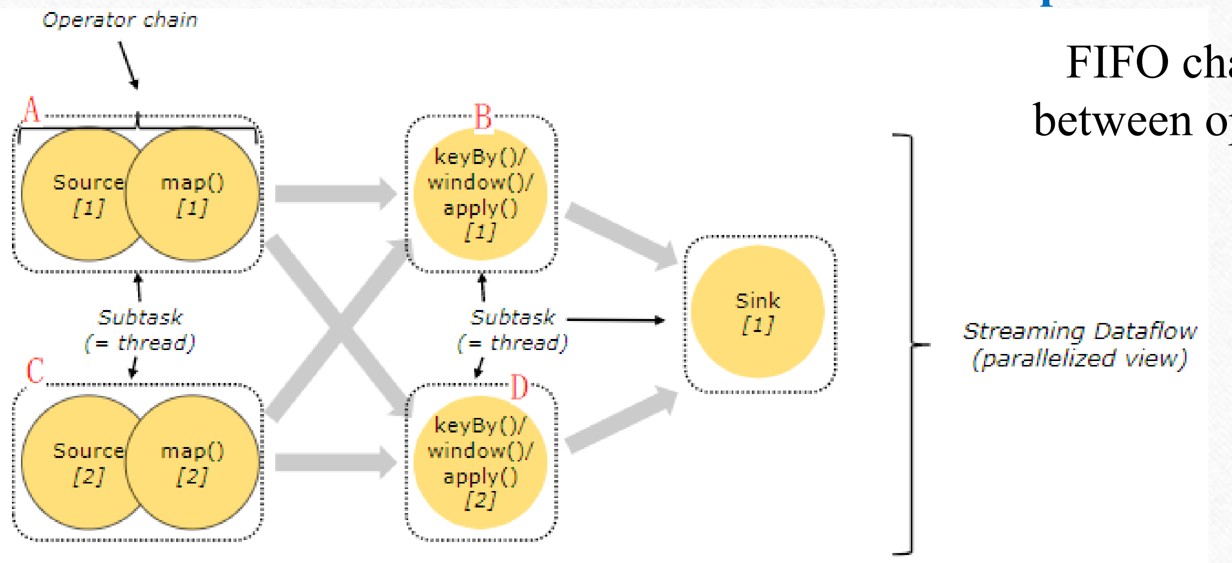


# Stream Processing Engines

- Framework aimed at facilitating the development and deplyment of DSP applications
- Mainly targeting scale-out platforms (e.g., clusters, clouds, cloud+edge)

**Operator = thread**

FIFO channels
between operators



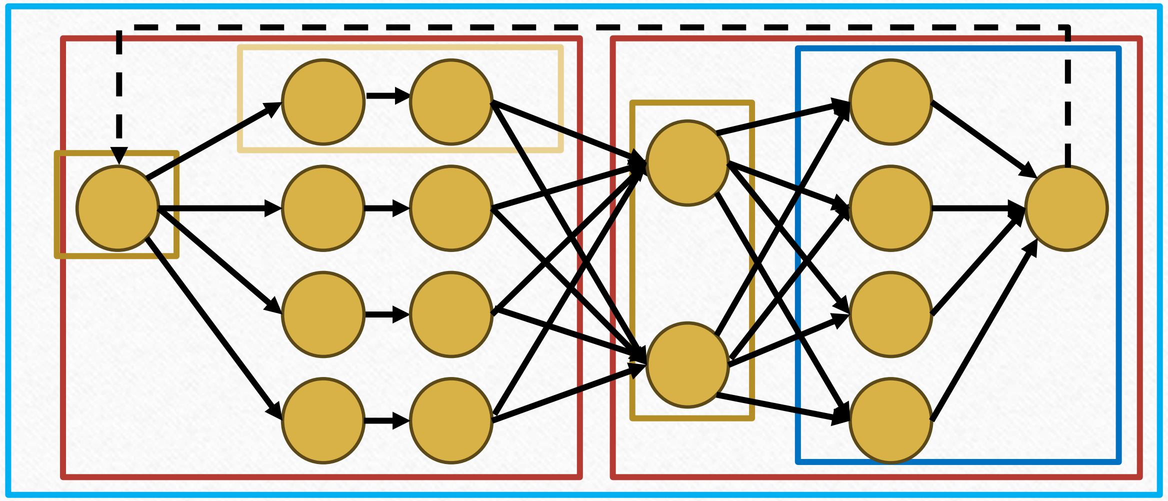
# Work motivations

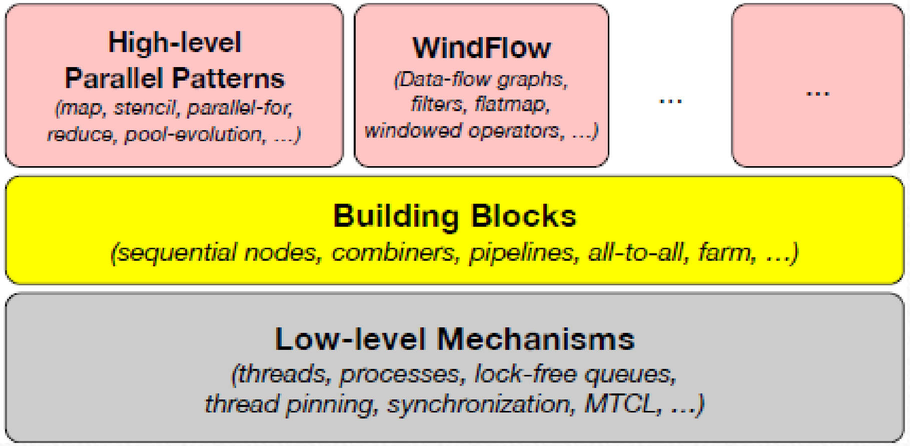
- **Challenges in DSP**
  - Many current stream processing frameworks are primarily designed for distributed systems and often rely on the JVM
  - These frameworks may not efficiently exploit the full capabilities of scale-up architectures (single machines equipped with multi-core CPUs and GPUs). The overhead introduced by the JVM and the lack of fine-grained control over operators/threads placements can lead to suboptimal performance
  - Efficient scale-up solutions are essential for maximizing resource utilization on powerful single-node systems, reducing latency, and improving throughput
- **Objective of our study**
  - Improve the performance of DSP applications on scale-up scenarios by leveraging C++-based frameworks (i.e., **WindFlow**)
  - OS scheduling may scatter communicating operators, increasing cache misses and thread migration. We explore the benefit of thread-to-core pinning strategies to optimize CPU cache usage and reduce context-switching overhead

# FastFlow – Stream Parallel Programming

- Header-only C++ library. It promotes data-flow streaming as a first-class concept to design parallel (and distributed) applications using a set of composable and nestable **building blocks**

A concurrent streaming network built by composing sequential and parallel building blocks

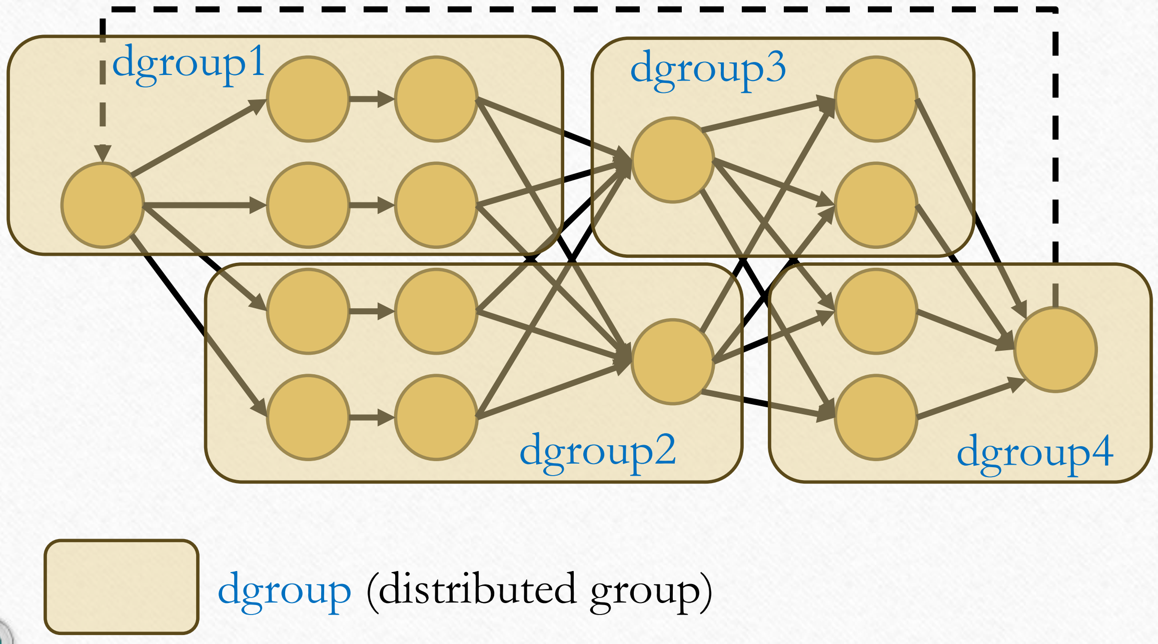


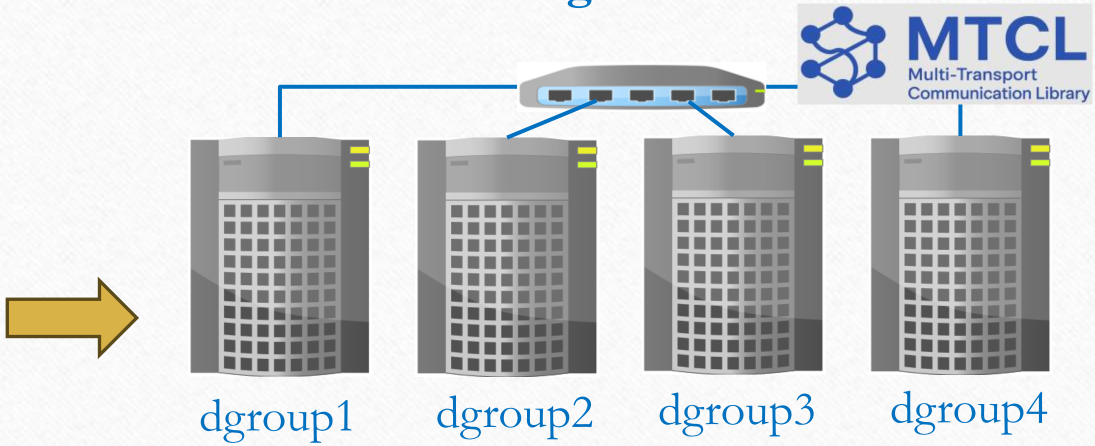



```
pipe( all2all( seq(1), pipe(4)( seq(1), seq(1) ),  all2all( seq(2), all2all( seq(4), seq(1) ) ) ).feedback();
```

# FastFlow – Stream Parallel Programming

- Header-only C++ library. It promotes data-flow streaming as a first-class concept to design parallel (and distributed) applications using a set of composable and nestable **building blocks**



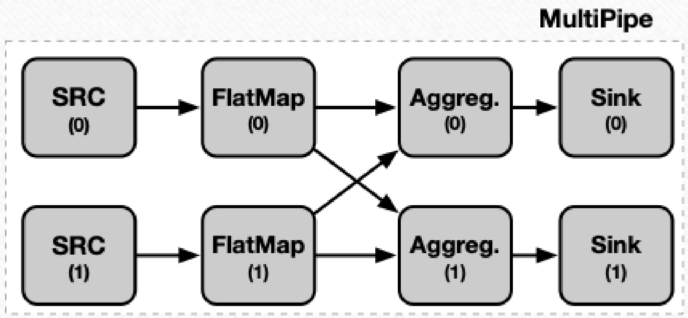

A dgroup is a plain shared-memory FastFlow application with an enhanced runtime system for message routing, serialization, and communication build on top of the MTCL library

# WindFlow – DSP on scale-up platforms

- **C++17 header-only library**. WindFlow uses FastFlow as a runtime system, leveraging building blocks
- It simplifies the development of efficient DSP applications on scale-up platforms (**CPUs + GPUs**)

```
PipeGraph app;
Source src = Source_Builder(…)
                 .withParallelism(2)
                 .build();

FlatMap fm = FlatMap_Builder([](input_t &t, Shipper<output_t> &s) {…}
                 .withParallelism(2)
                 .build();

Aggregation ag = Aggregation_Builder([](const output_t &t1,
                                        const output_t &t2) -> output_t {…})
                 .withParallelism(2)
                 .withKeyBy([](const output_t &t) -> key_t {…})
                 .withTimeBoundaries(seconds(10), seconds(1))
                 .build();

Sink snk = Sink_Builder(…)
                 .withParallelism(2)
                 .build();

app.add_source(src).add(fm).add(ag).add_sink(snk);
```

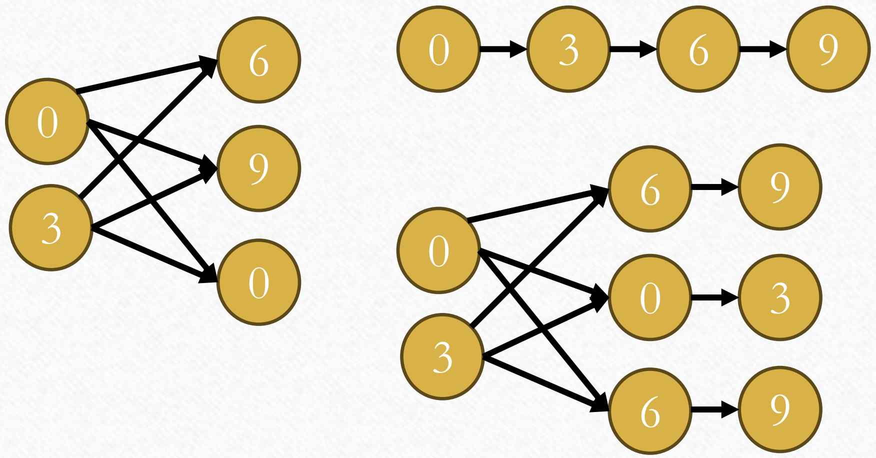
**Fluent interface** to declare and configure operators



# Thread-to-core pinning in FastFlow/WindFlow

- **Thread pinning**, i.e., binding threads to cores, reduces context switches and enhances cache locality, especially on NUMA or chiplet CPUs
- FastFlow provides thread-to-core affinity through a statically defined configuration file that contains a string reporting the actual order of cores in the system. Threads are then assigned to a single core using a round-robin policy on the listed cores
- Limitations:
  - It is challenging to assign specific threads to cores in complex data-flow streaming graph
  - The user may manually map threads to cores by using the FastFlow's APIs (i.e., ff_mapThreadToCPU())
- WindFlow inherits FastFlow's limitations for thread affinity

**FF_MAPPING_STRING**="0, 3, 6, 9"



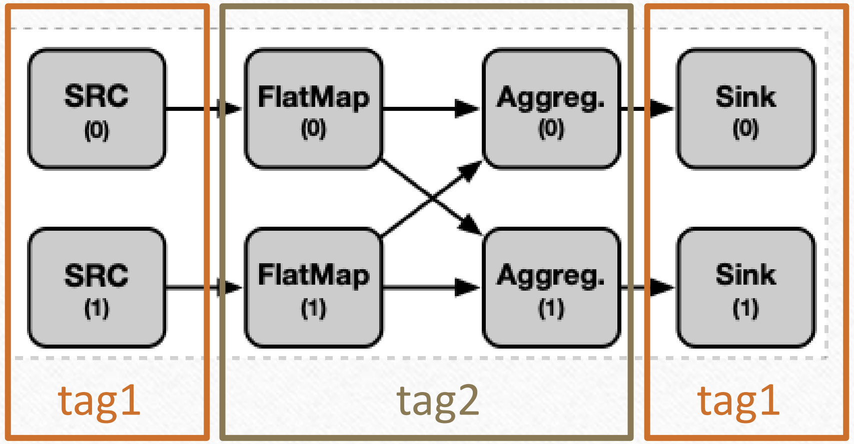
# New Thread-to-core pinning in FastFlow

- We improved the **FastFlow** affinity low-level mechanisms
- We introduced affinity APIs in **WindFlow**'s fluent-based interface

- **FastFlow** now provides an FF_AFFINITY environmental variable with a more complex and flexible syntax.
  - FF_AFFINITY="[0:8:2, 1]" → CPU-set: {0,2,4,6,8,10,12,14,1} defines an **anonymous** CPU-set
  - FF_AFFINITY="**even**[0:8:2], **odd**[1:8:2]" → 2 **labelled** CPU-sets: even{0,2,4,6,8,10,12,14} odd{1,3,5,7,9,11,13,15}
- CPU set may have an **assignment policy**: currently, we have only **rr** (round-robin)
- **FastFlow** nodes can be dynamically associated with labels using set_affinity_tag()/get_affinity_tag()

# New Thread-to-core pinning in FastFlow

- Suppose we define FF_AFFINITY="tag1[0:4:1]rr, tag2[5:4:1]" for executing our example application
- This string will produce 5 CPU sets for the FastFlow nodes labelled with tag1 and tag2
- Nodes labelled with tag1 will be executed each on a single core of the set 0,1,2,3 using a round-robin (rr) assignmentent
  - SRC(0) → {0} SRC(1) → {1} Sink(0) → {2} Sink(1) → {3}
  - In this case, threads labeled with tag1 are pinned to a single core
- Nodes tagged with tag2, will be executed on all cores of the single CPU set {5,6,7,8}
  - FlatMap(0), FlatMap(1), Aggreg(0), Aggreg(1) → {5,6,7,8}
  - In this case, the OS could freely move them among the cores of the set



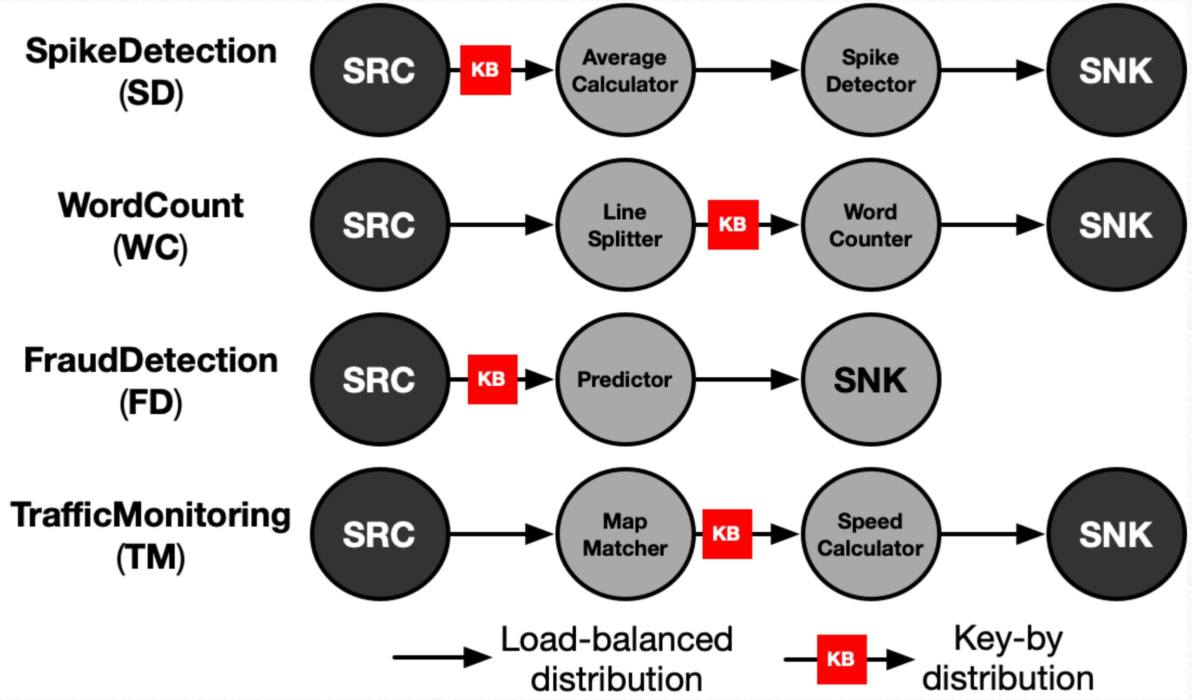
# Operator pinning in WindFlow

- Pinning choices can be specified at the **WindFlow** level
- Programmers can assign one or more **labels** to each operator, and these labels must match the **tags** defined in the **FF_AFFINITY** environment variable
- When a single label is provided, all replicas of the operator, along with their underlying threads, are assigned to the same CPU set associated with the tag
- Alternatively, different labels can be assigned to each replica individually, or a list of labels can be provided and distributed among replicas using a round-robin assignment

```
PipeGraph app;
Source src = Source_Builder(…)
                  .withParallelism(2)
                  .withPinning("odd")
                  .build();

FlatMap fm = FlatMap_Builder([](input_t &t, Shipper<output_t> &s)
{…}
                  .withParallelism(2)
                  .withPinning("odd | even")
                  .build();

Aggregation ag = Aggregation_Builder([](const output_t &t1,
                                         const output_t &t2) {…})
                  .withParallelism(2)
                  .withKeyBy([](const output_t &t) -> key_t {…})
                  .withTimeBoundaries(seconds(10), seconds(1))
                  .withPinning("odd | even")
                  .build();

Sink snk = Sink_Builder(…)
                  .withParallelism(2)
                  .withPinning("even")
                  .build();
```

# Benchmarks used for the tests

- Benchmark applications from the **DSPBench** benchmark suite



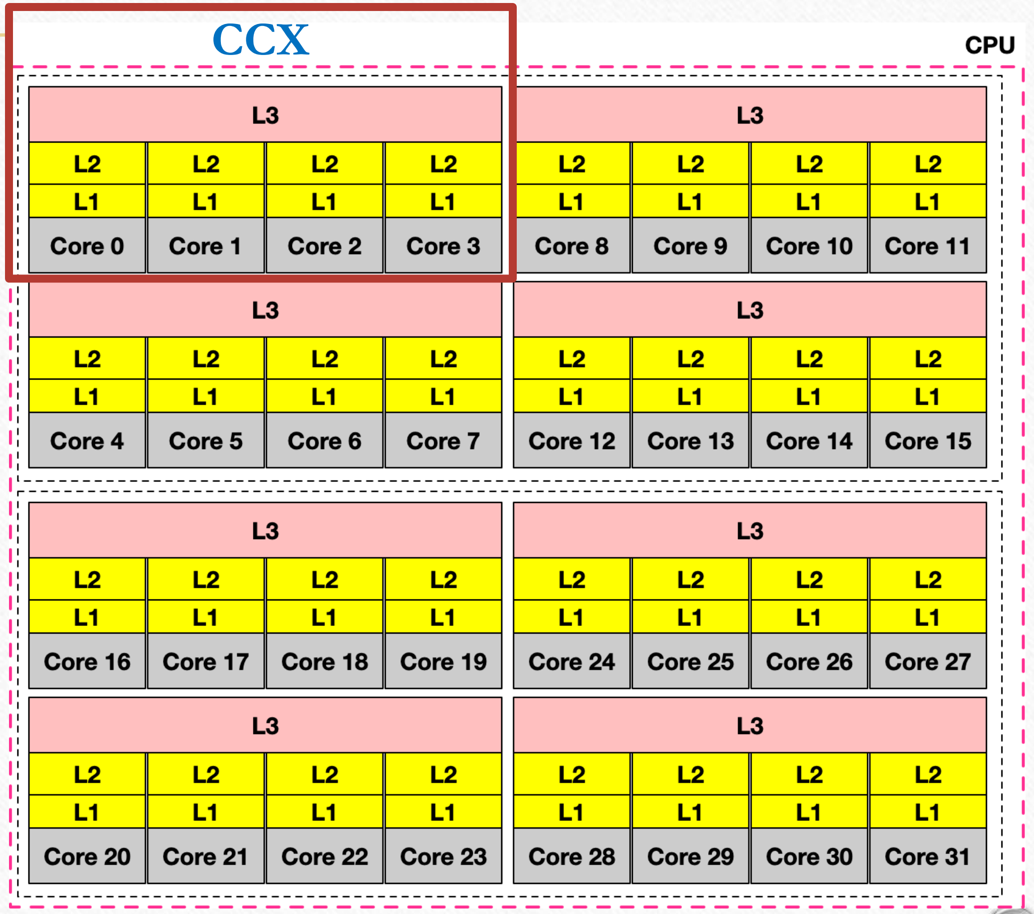

**Logical data-flow graphs**

Windflow implementation:

- A PipeGraph where each operator is parallelized with a given parallelism degree (# replicas)
- Each replica runs on a dedicated FastFlow thread
- Consecutive operators can be combined to reduce the number of threads and coarsen the thread granularity (not done for our tests)
- KB implemented with an all2all building block

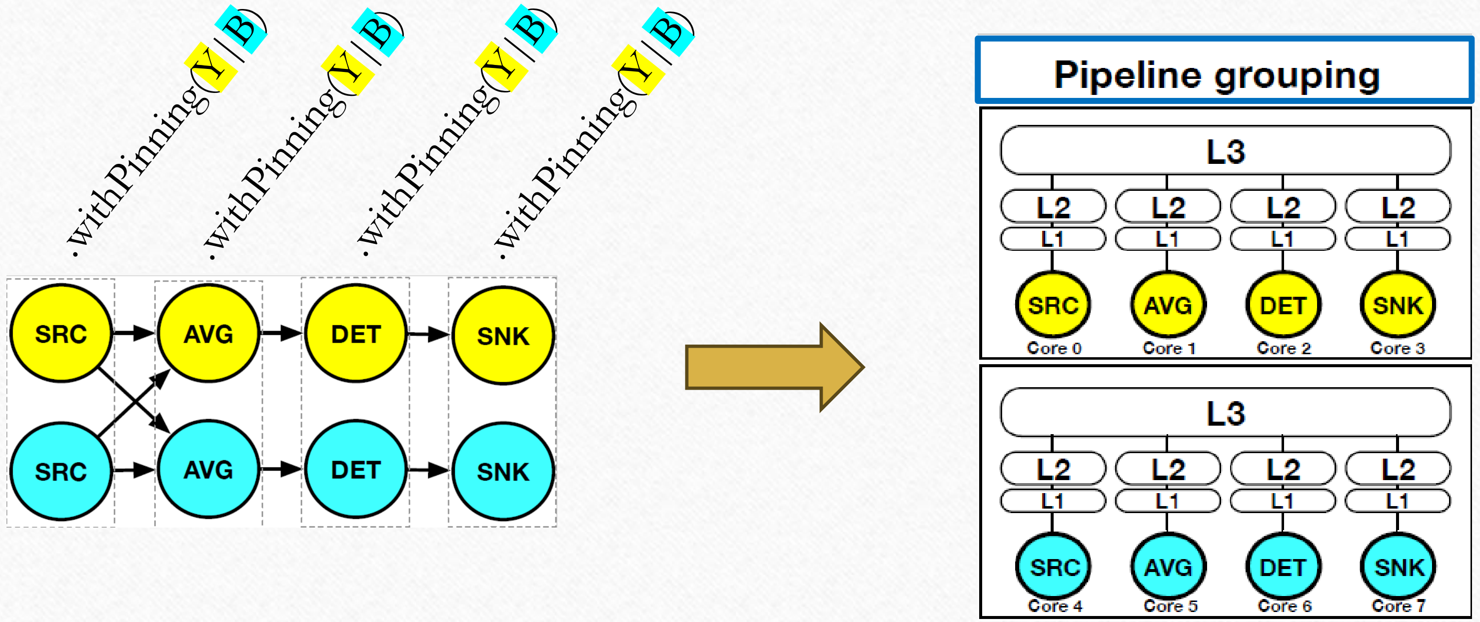
# Target Architecture & Performance Metrics

- **2 AMD EPYC 7551 CPU (2 GHz)**
  - Each CPU has four dies, each with two core clusters (**CCX**). Each CCX has four physical cores (two thread contexts each) with private L1d/L2 caches (64 KiB and 512 KiB) and a shared L3 cache (8 MiB)
  - Cores in different CCXs, even within the same die, do not share any cache levels.
  - Total number of physical cores: 64 on two CPUs
  - **In our test we considered only one CPU**
- **Performance metrics considered**
  - Throughput in tuples/sec
  - L3 accesses/misses



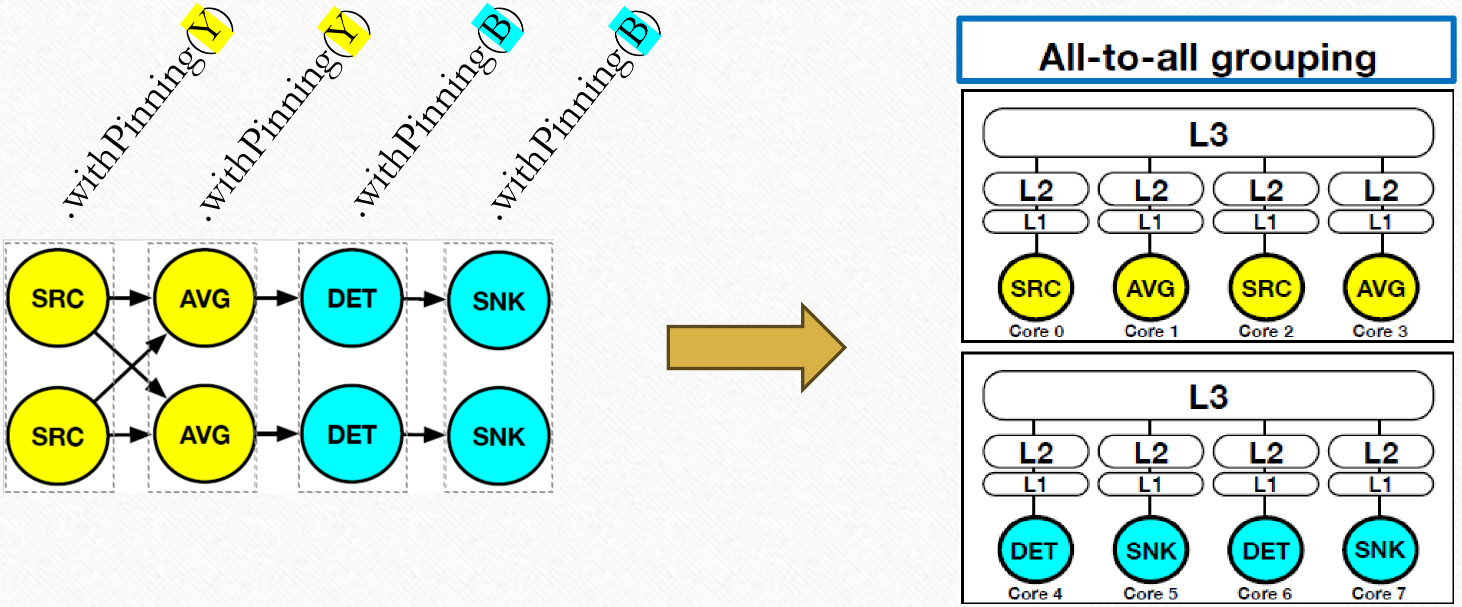
# Pinning Strategies 1/6

FF_AFFINITY="Y[0:4]rr, B[4:4]rr"



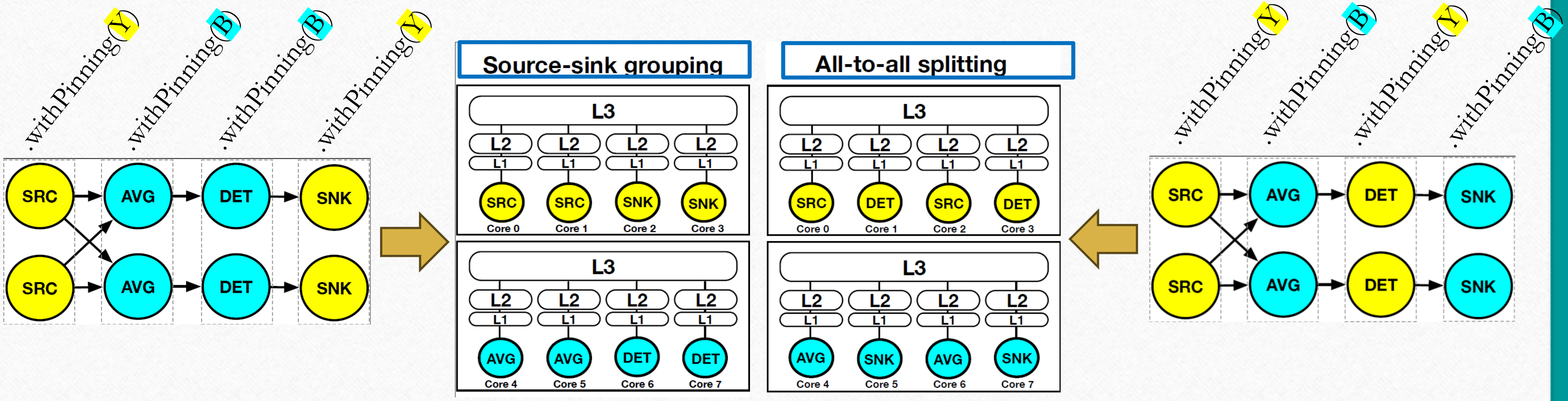
# Pinning Strategies 2/6

FF_AFFINITY="Y[0:4]rr, B[4:4]rr"

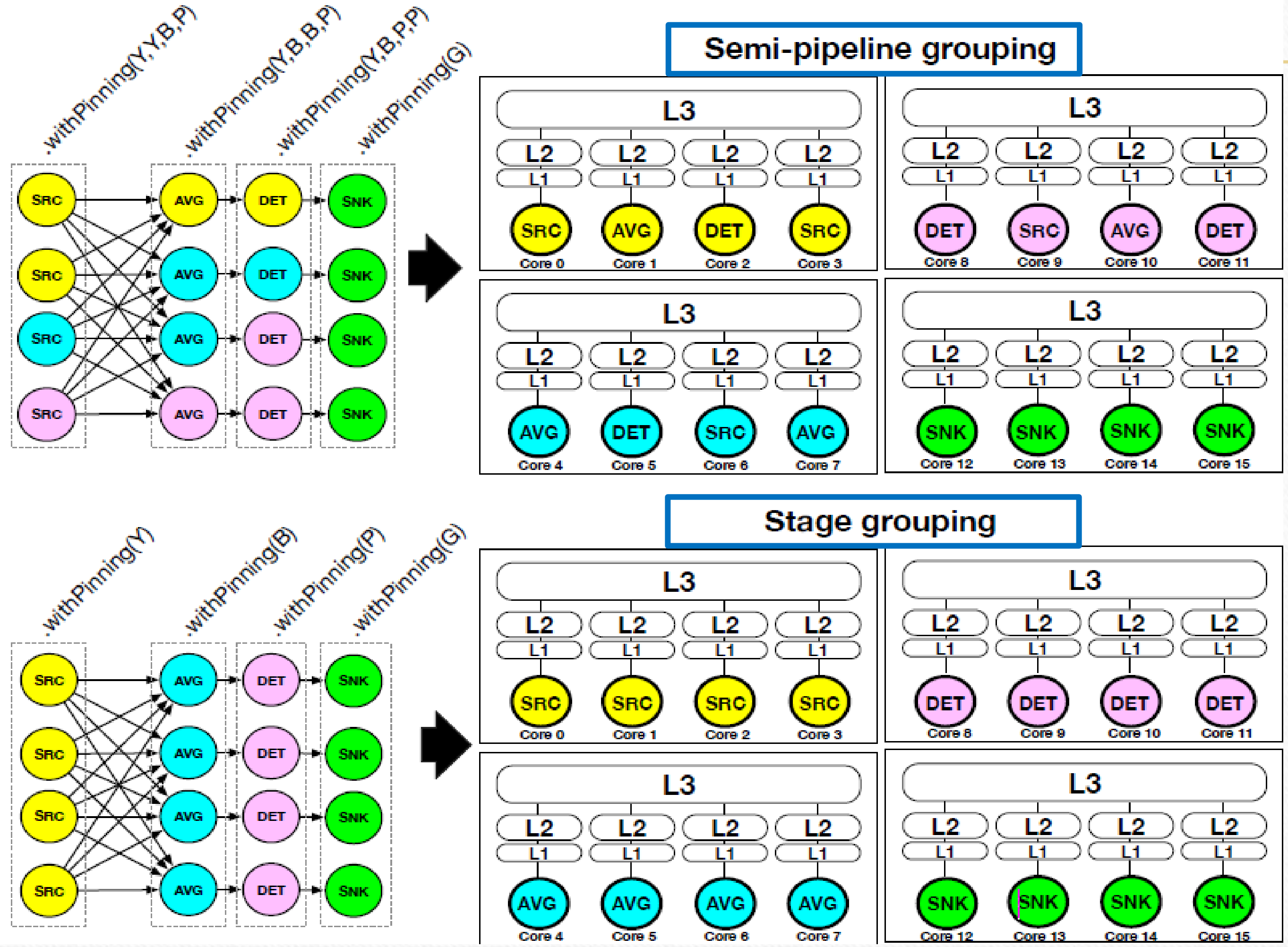


# Pinning Strategies 3-4/6

FF_AFFINITY="Y[0:4]rr, B[4:4]rr"

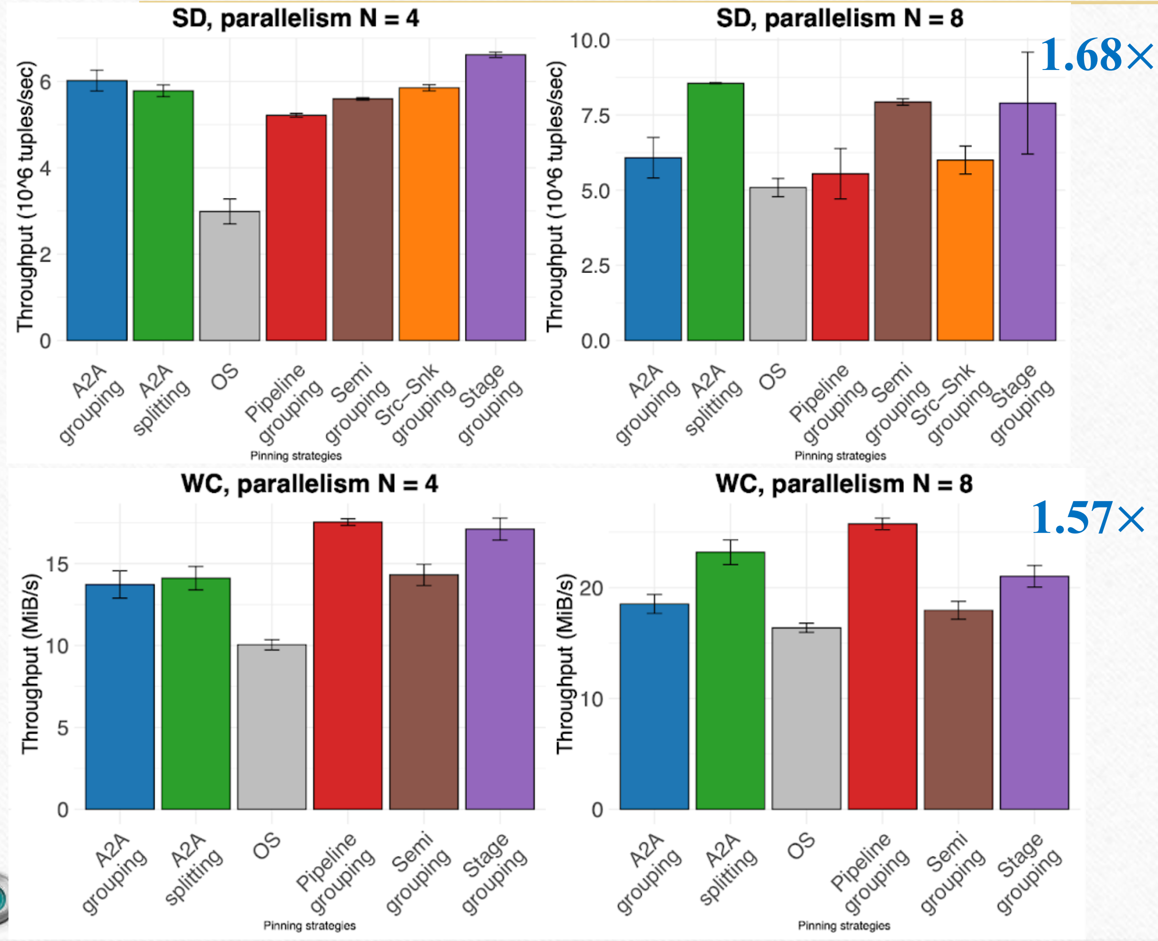


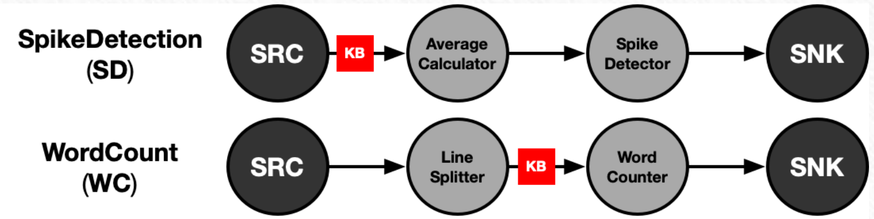
# Pinning Strategies 5-6/6




FF_AFFINITY="Y[0:4]rr,
B[4:4]rr,
P[8:4]rr,
G[12:4]rr,"

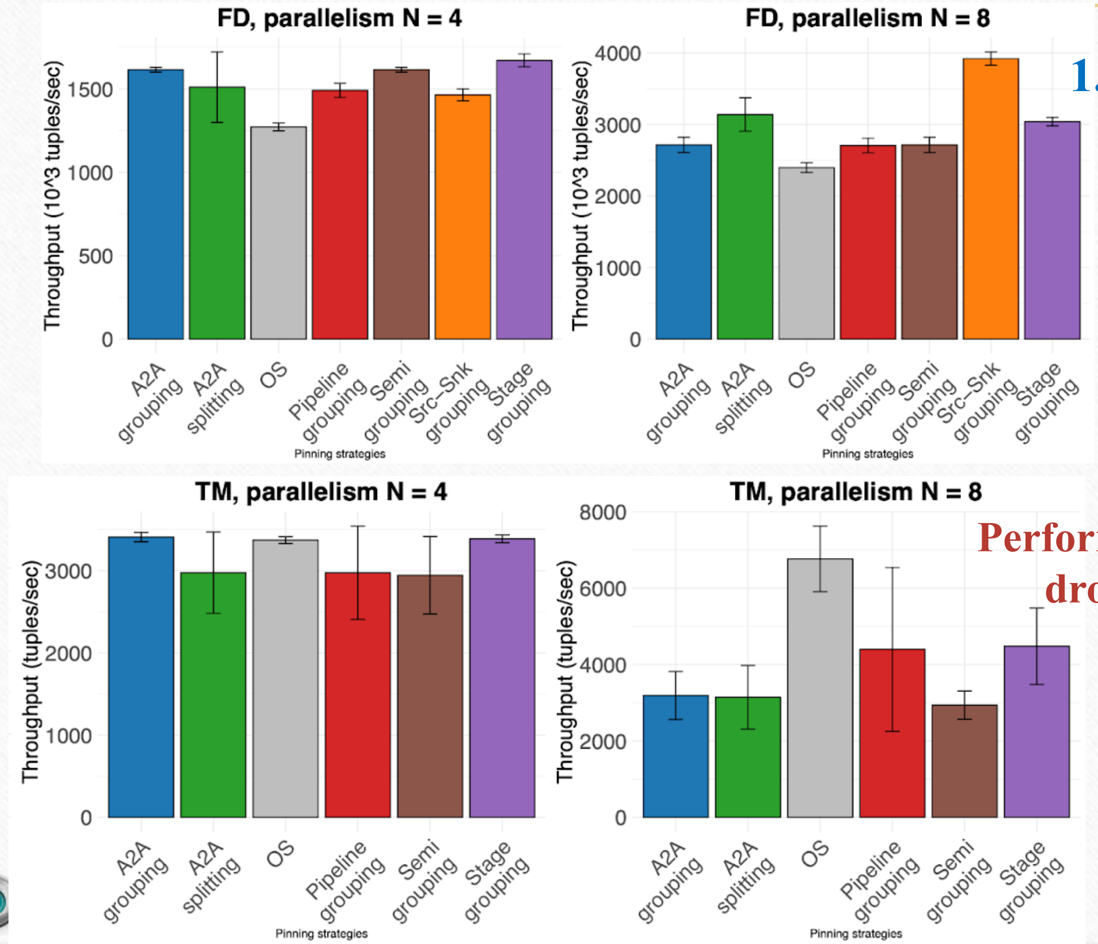
# SD and WC Applications



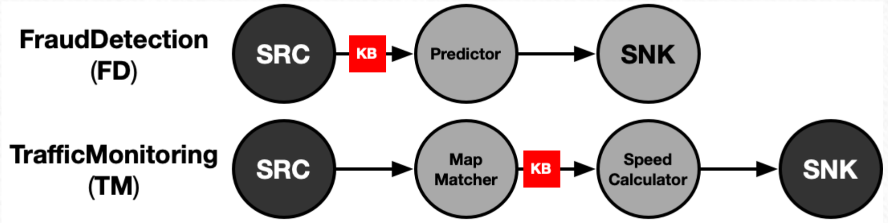

- All strategies perform consistently better than the default OS (i.e., no thread-to-core pinning)
- For SD, there is no clear winner overall
- With increasing parallelism degree for the operators, the average improvement decreases
- For WC, the best strategy is Pipeline grouping

# FD and TM Applications



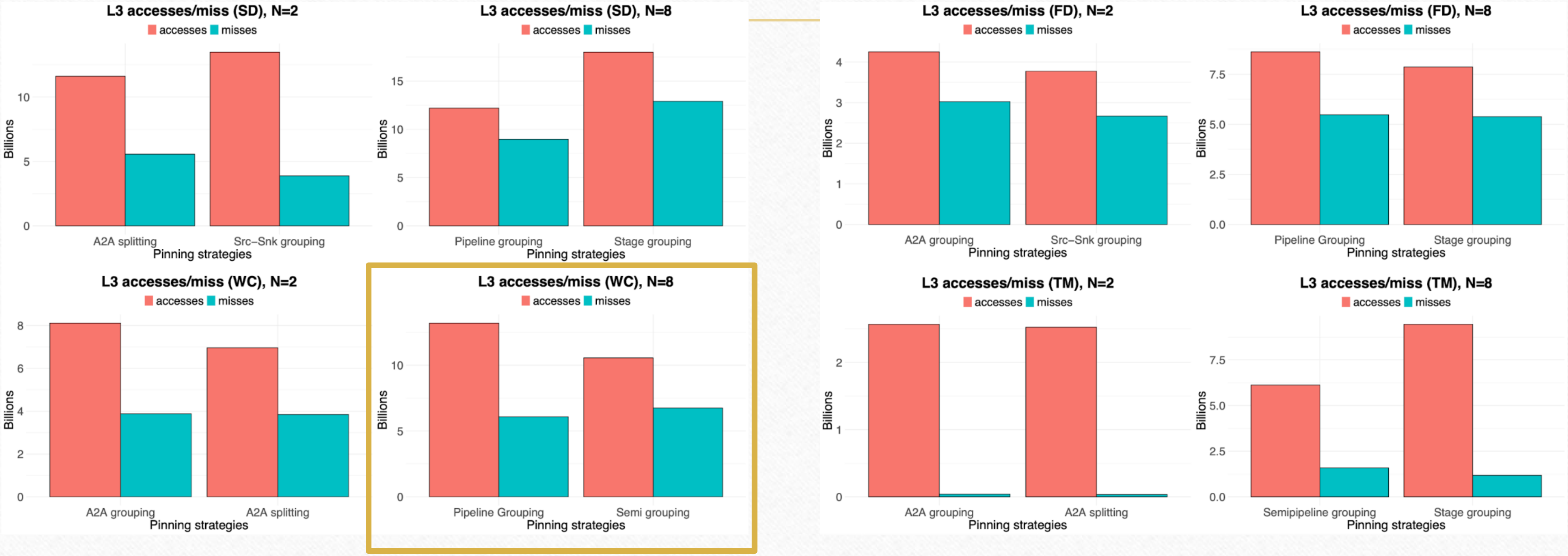

1.64 ×

Performance drop!




- For FD, all strategies perform consistently better than the default OS. Source-Sink grouping results in the best strategy with high parallelism
- For TM, the no-pinning strategy (OS) is the best policy to adopt
- TM has the lowest throughput among all benchmarks, and it uses an external library for geo-localization, making it more coarse-grained

# Cache Profiling




- The cache profiling analysis showed that the increased throughput is primarily due to a lower LLC miss rate
- This was confirmed in 24 out of 32 tests we executed. In the remaining tests, the differences are negligible.

# Outcomes

- In many cases (though not all), thread-to-core pinning provides a performance boost in DSP applications
  - This is particularly true for fine-grained and high-throughput applications
- The performance gains mostly come from a more efficient use of the memory hierarchy
- Finding the best thread-to-core pinning assignment for a given architecture is a non-trivial task
  - Difficult to automate this process in a portable way
- We advocate a "trial and error approach". Therefore, providing flexible tools for experimenting with different strategies is essential to maximize performance

# Conclusions and Future Work

- **What we have done**
  - We addressed the problem of thread pinning for DSP applications in **WindFlow**
  - We enhanced the thread affinity mechanisms in **FastFlow** and exposed a suitable API in **WindFlow**
  - Experimental analysis demonstrated the effectiveness of pinning in most cases and the importance of leveraging flexible tools
- **Plans for the future**
  - Extend the performance analysis using more benchmarks
  - Consider different NUMA architectures
  - Exploring the impact of hyperthreading and multiple CPU sockets
  - Further extend the FF_AFFINITY assignment policies

Thank you!