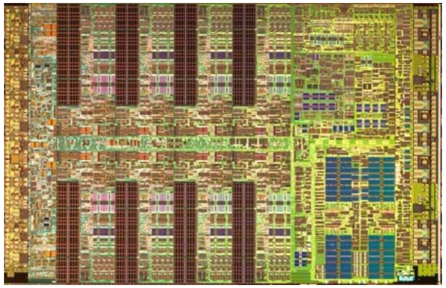
# Flexible Filters for High Performance Embedded Computing

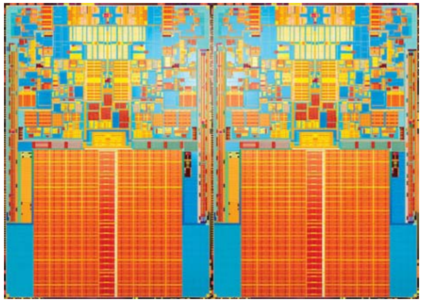
Rebecca Collins and Luca Carloni
Department of Computer Science
Columbia University

# Motivation

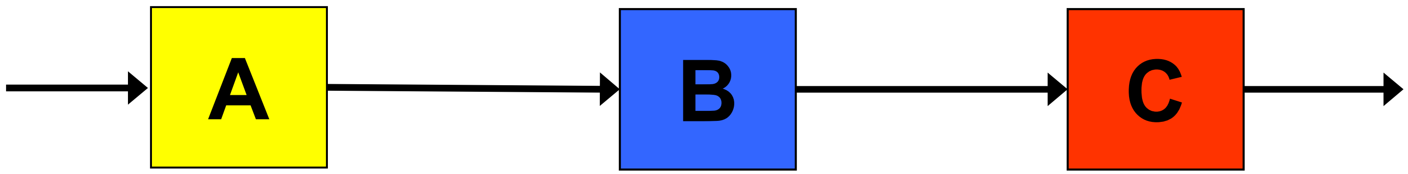

Sony Ericsson


Call Log
Phonebook
Message Box
Touch Player
Camera
FM Radio
Internet
Photo Slides
Media Album
Clock
Programs
Settings
Shortcuts
Exit
SAMSUNG



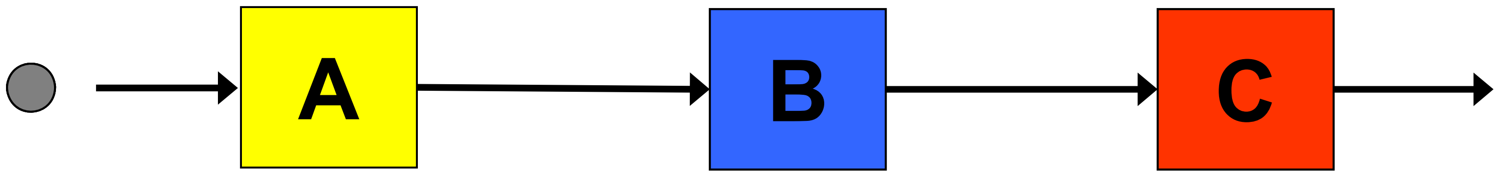


# Stream Programming



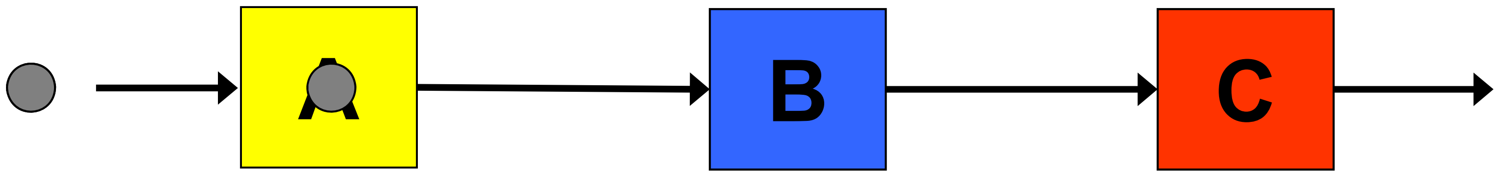

- **Stream Programming model**
  - filter**: a piece of sequential programming**
  - channels**: how filters communicate**
  - token**: an indivisible unit of data for a filter**
- **Examples: Signal processing, image processing, embedded applications**

# Stream Programming



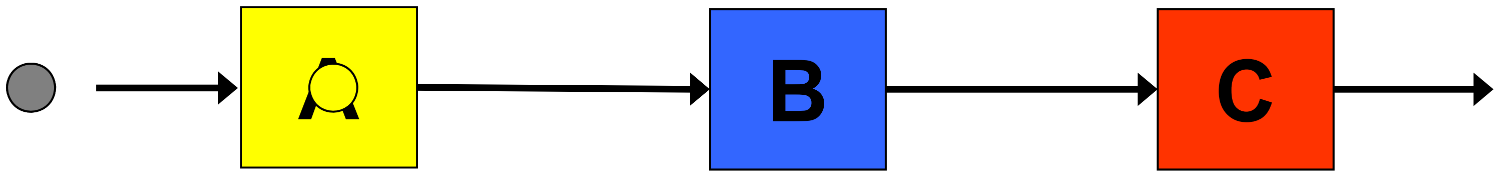

- **Stream Programming model**
  - filter**: a piece of sequential programming**
  - channels**: how filters communicate**
  - token**: an indivisible unit of data for a filter**
- **Examples: Signal processing, image processing, embedded applications**

# Stream Programming



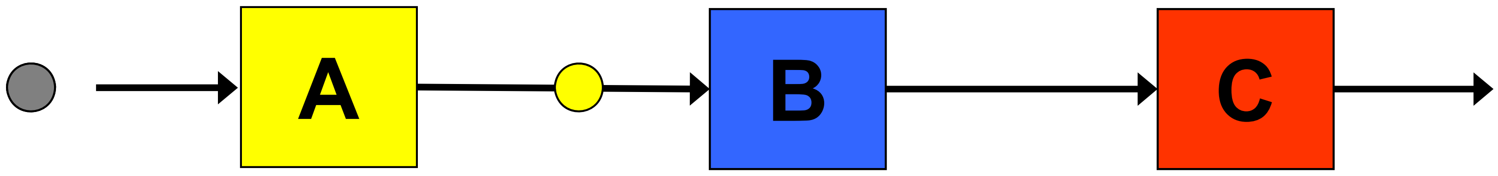

- **Stream Programming model**
  - filter**: a piece of sequential programming**
  - channels**: how filters communicate**
  - token**: an indivisible unit of data for a filter**
- **Examples: Signal processing, image processing, embedded applications**

# Stream Programming



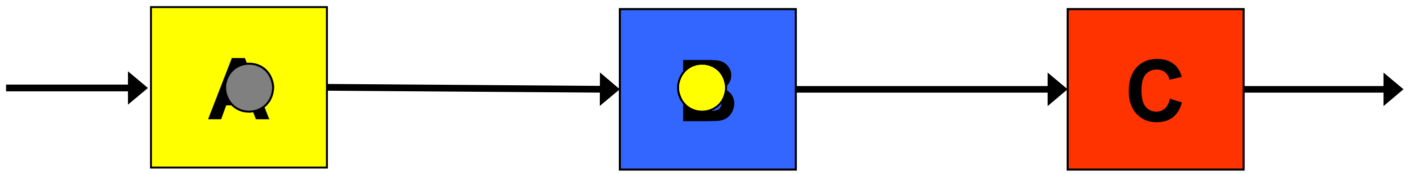

- **Stream Programming model**
  - filter**: a piece of sequential programming**
  - channels**: how filters communicate**
  - token**: an indivisible unit of data for a filter**
- **Examples: Signal processing, image processing, embedded applications**

# Stream Programming



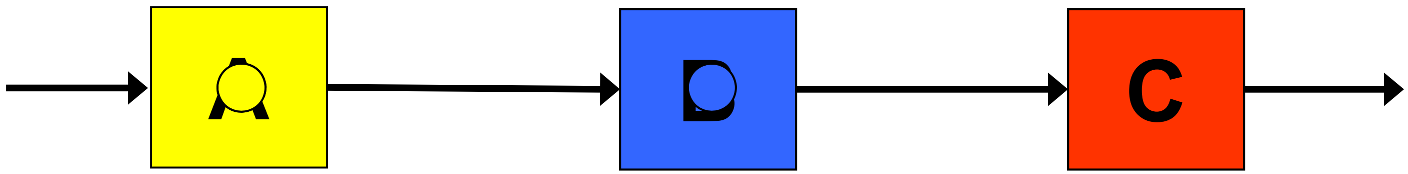

- **Stream Programming model**
  - filter: **a piece of sequential programming**
  - channels: **how filters communicate**
  - token: **an indivisible unit of data for a filter**
- **Examples: Signal processing, image processing, embedded applications**

# Stream Programming



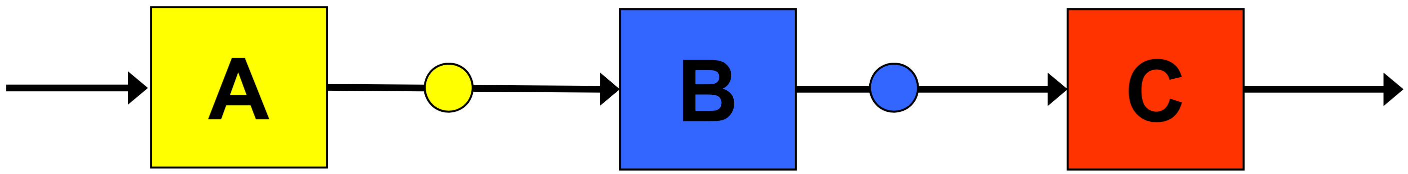

- **Stream Programming model**
  - filter**: a piece of sequential programming**
  - channels**: how filters communicate**
  - token**: an indivisible unit of data for a filter**
- **Examples: Signal processing, image processing, embedded applications**

# Stream Programming



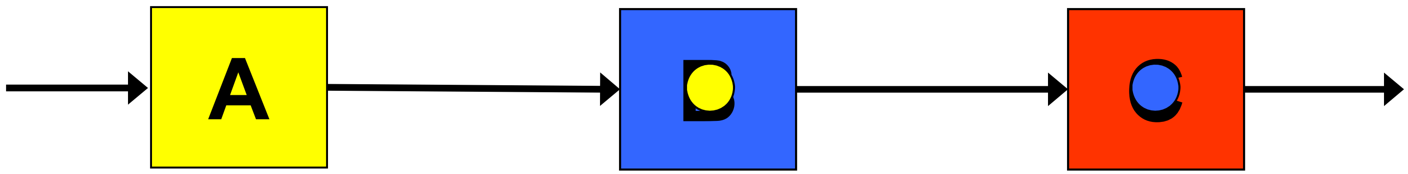

- **Stream Programming model**
  - filter**: a piece of sequential programming**
  - channels**: how filters communicate**
  - token**: an indivisible unit of data for a filter**
- **Examples: Signal processing, image processing, embedded applications**

# Stream Programming



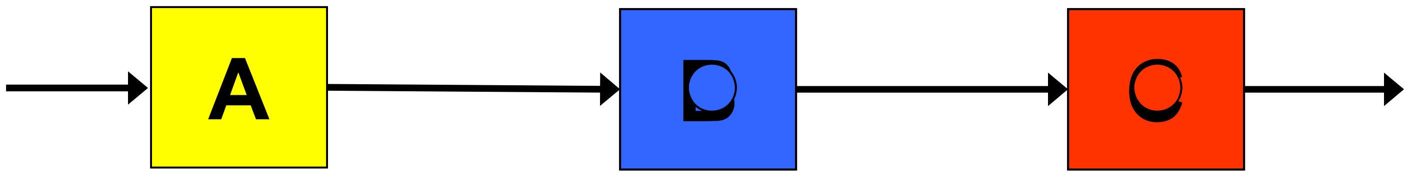

- **Stream Programming model**
  - filter: **a piece of sequential programming**
  - channels: **how filters communicate**
  - token: **an indivisible unit of data for a filter**
- **Examples: Signal processing, image processing, embedded applications**

# Stream Programming



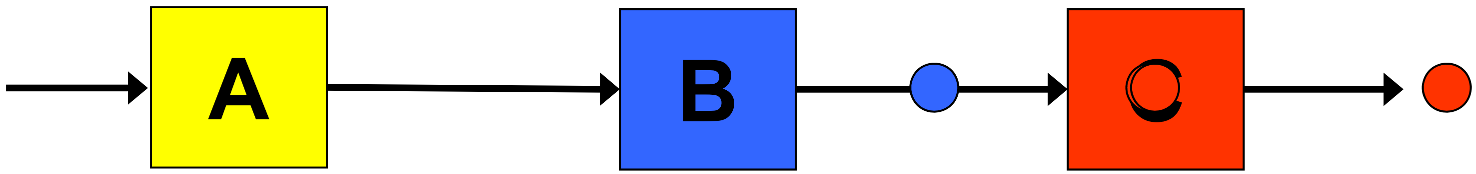

- **Stream Programming model**
  - filter: **a piece of sequential programming**
  - channels: **how filters communicate**
  - token: **an indivisible unit of data for a filter**
- **Examples: Signal processing, image processing, embedded applications**

# Stream Programming



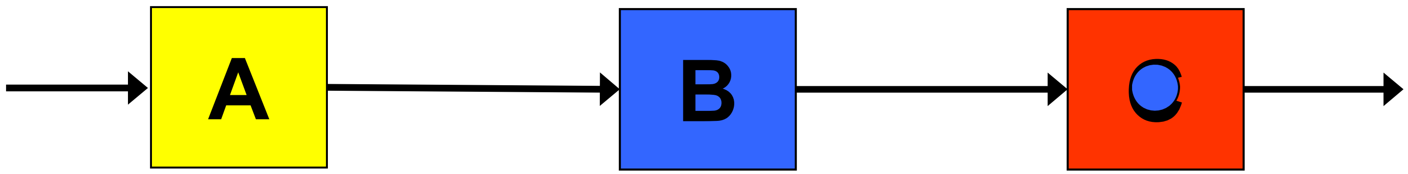

- **Stream Programming model**
  - filter**: a piece of sequential programming**
  - channels**: how filters communicate**
  - token**: an indivisible unit of data for a filter**
- **Examples: Signal processing, image processing, embedded applications**

# Stream Programming



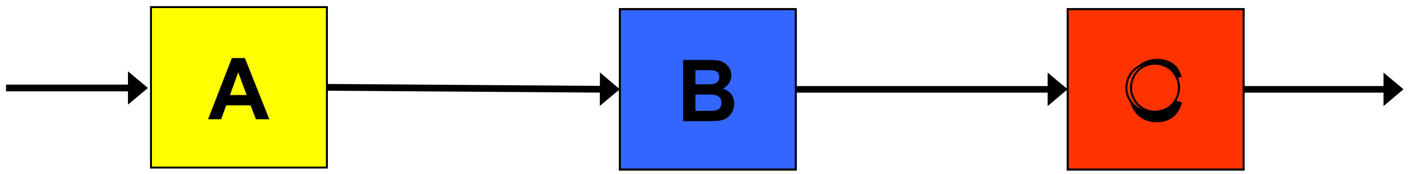

- **Stream Programming model**
  - filter**: a piece of sequential programming**
  - channels**: how filters communicate**
  - token**: an indivisible unit of data for a filter**
- **Examples: Signal processing, image processing, embedded applications**

# Stream Programming



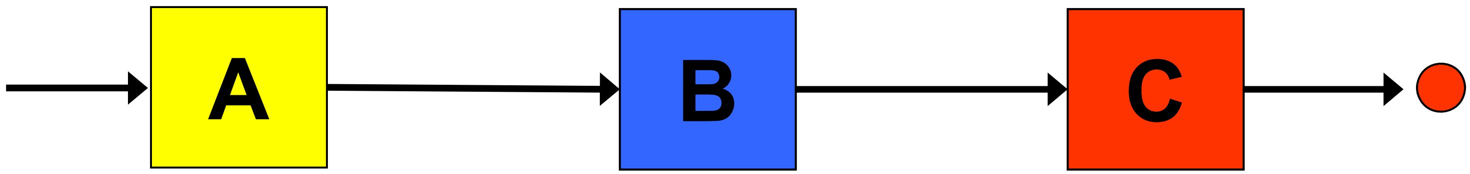

- **Stream Programming model**
  - filter: **a piece of sequential programming**
  - channels: **how filters communicate**
  - token: **an indivisible unit of data for a filter**
- **Examples: Signal processing, image processing, embedded applications**

# Stream Programming



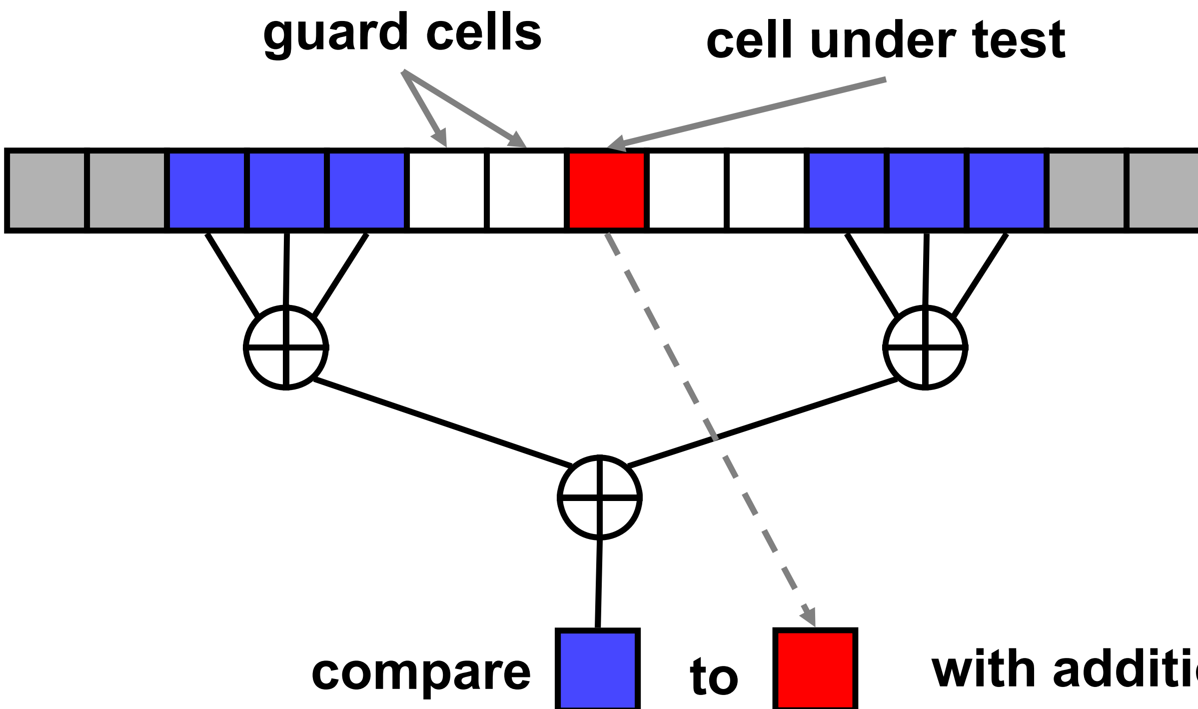

- **Stream Programming model**
  - filter: **a piece of sequential programming**
  - channels: **how filters communicate**
  - token: **an indivisible unit of data for a filter**
- **Examples: Signal processing, image processing, embedded applications**

# Stream Programming




- **Stream Programming model**
  - filter**: a piece of sequential programming**
  - channels**: how filters communicate**
  - token**: an indivisible unit of data for a filter**
- **Examples: Signal processing, image processing, embedded applications**

# Example: Constant False-Alarm Rate (CFAR) Detection

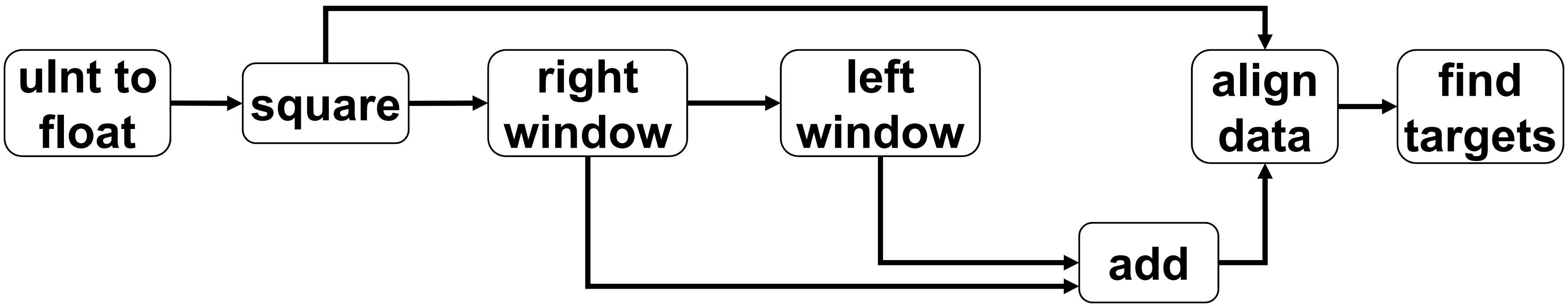



with additional factors
- number of gates
- threshold (μ)
- number of guard cells
- rows and other dimensional data

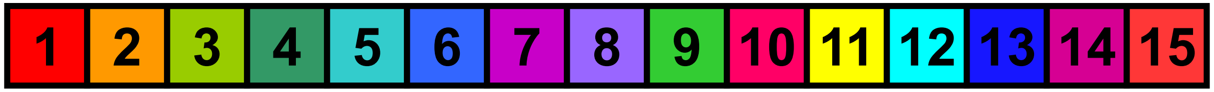
**HPEC Challenge**

http://www.ll.mit.edu/HPECchallenge/

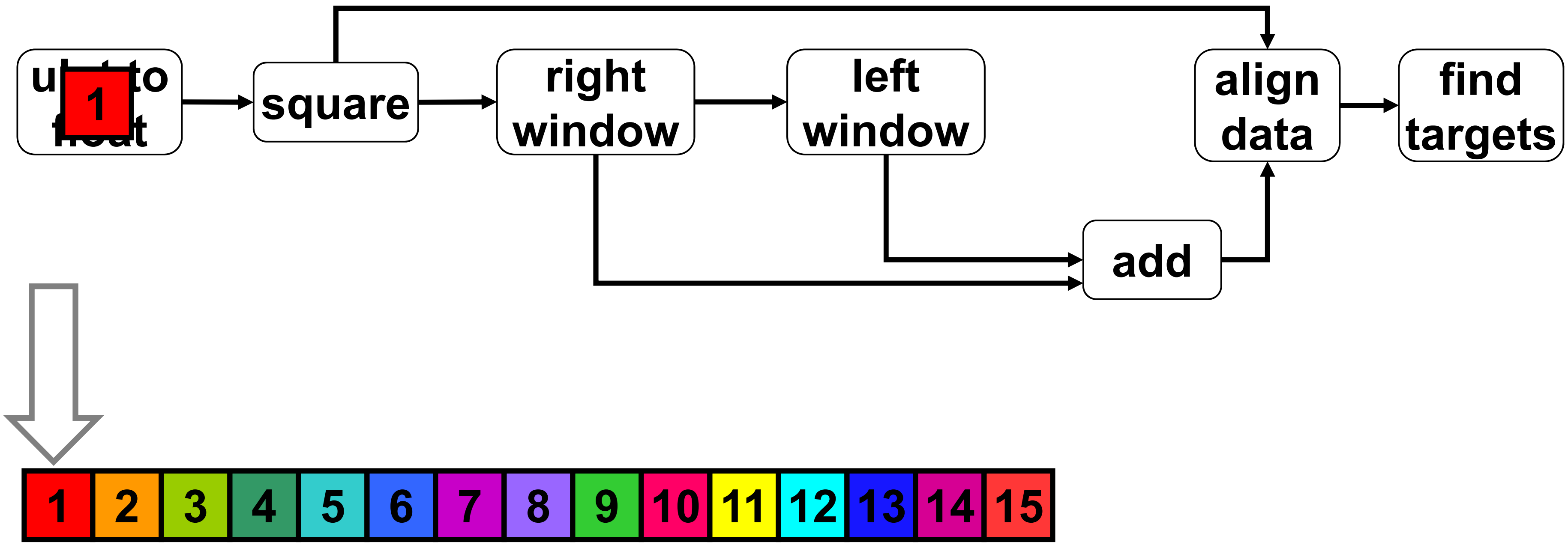
# CFAR Benchmark



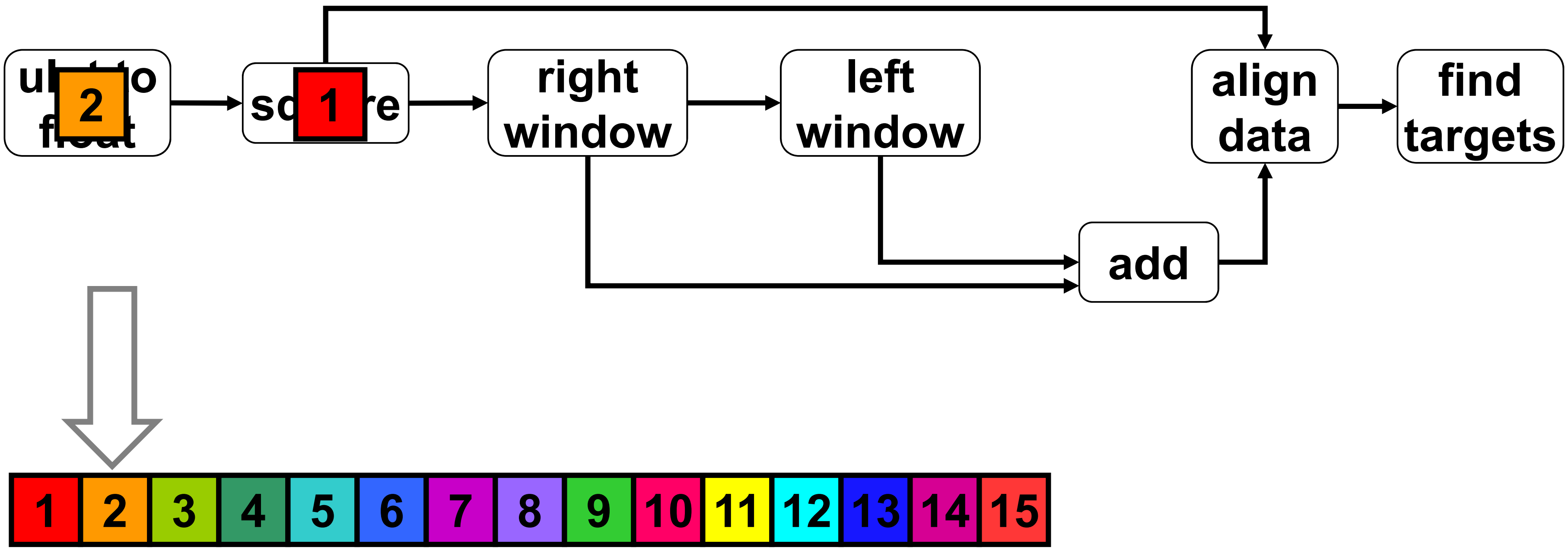

**Data Stream:**



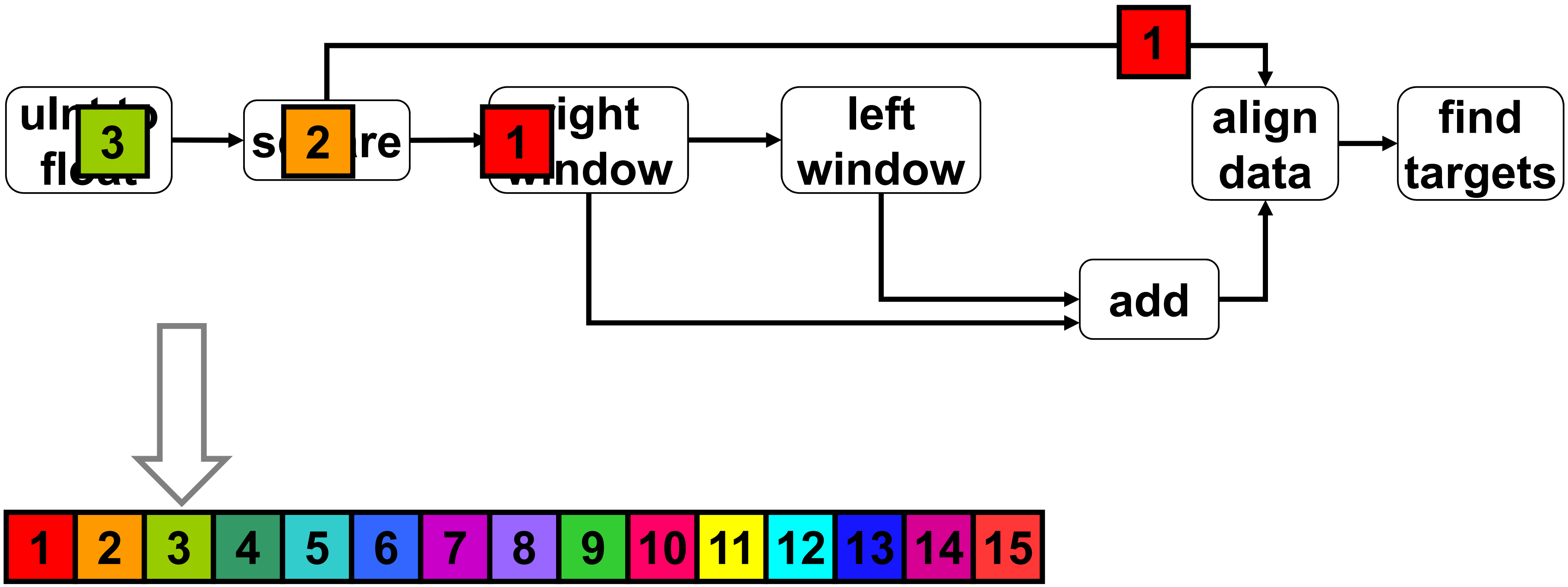
# CFAR Benchmark



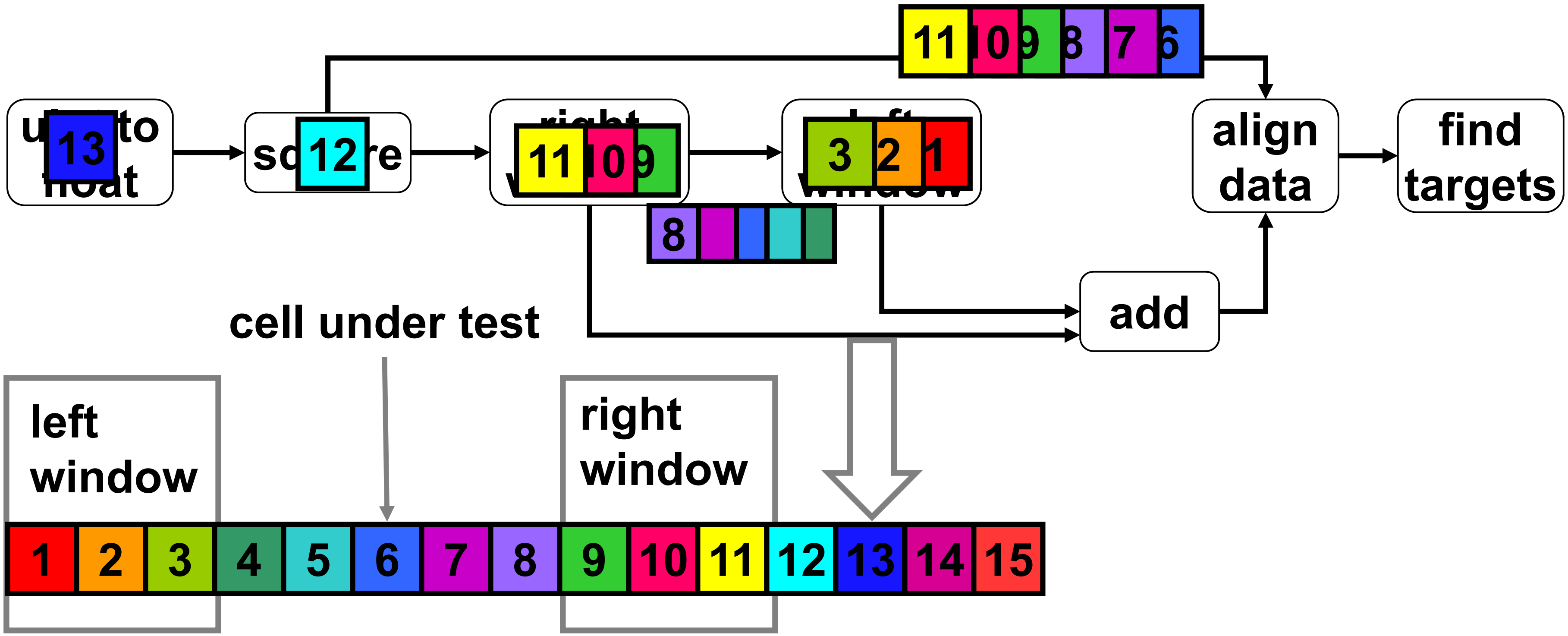
# CFAR Benchmark



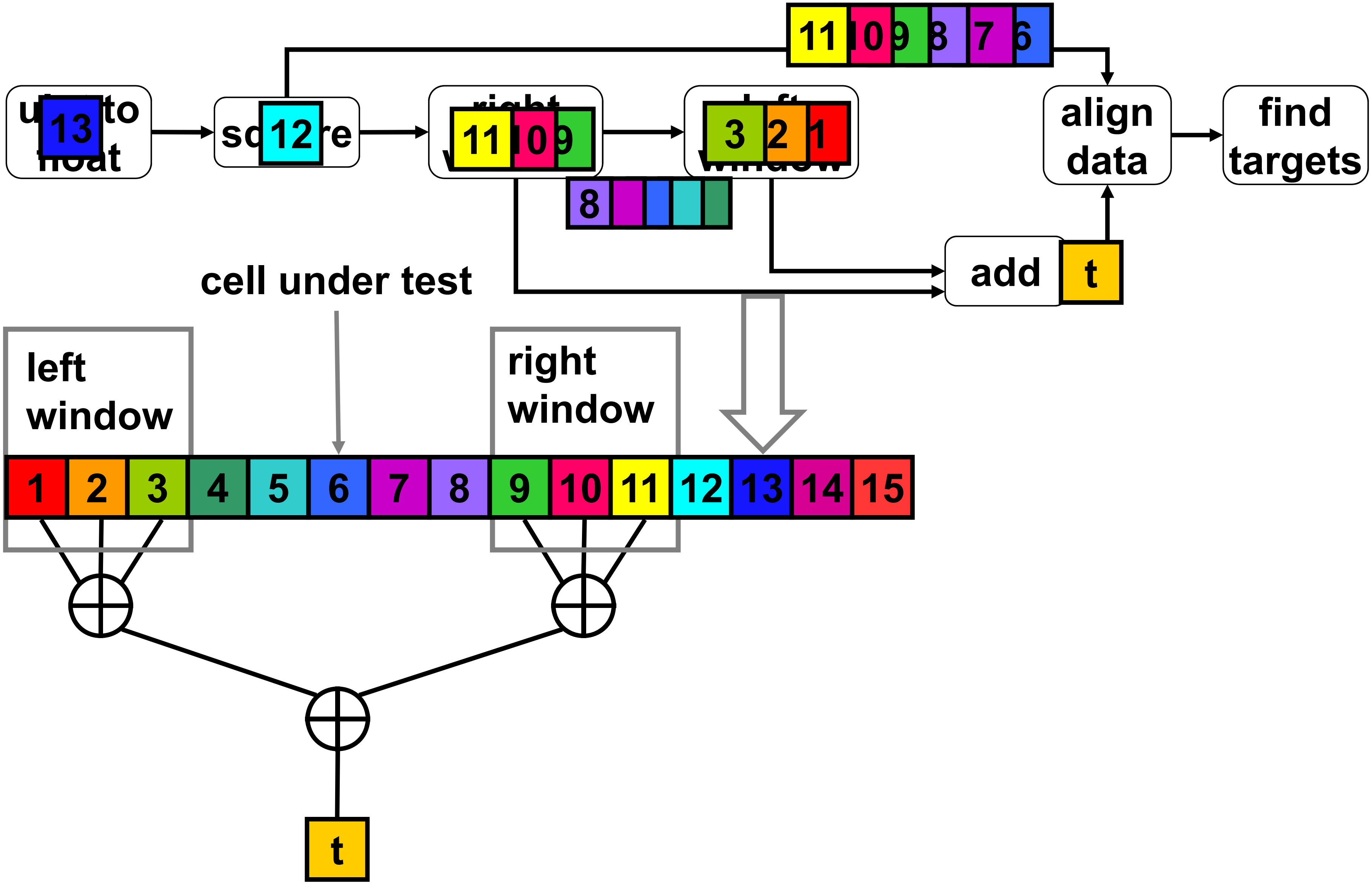
# CFAR Benchmark



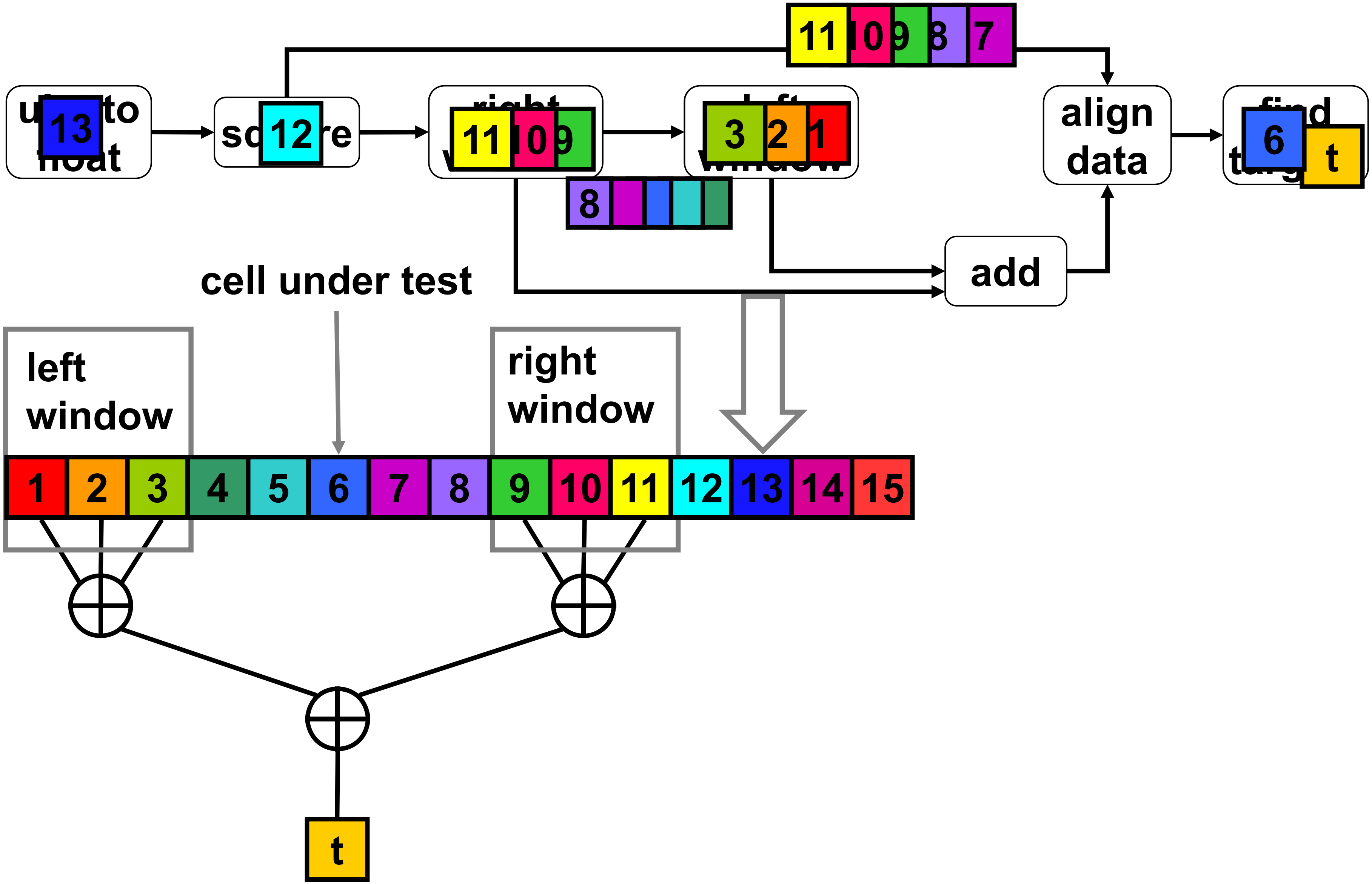
# CFAR Benchmark



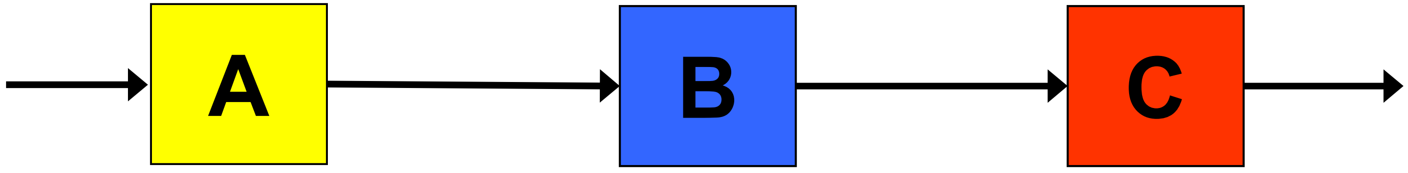
# CFAR Benchmark



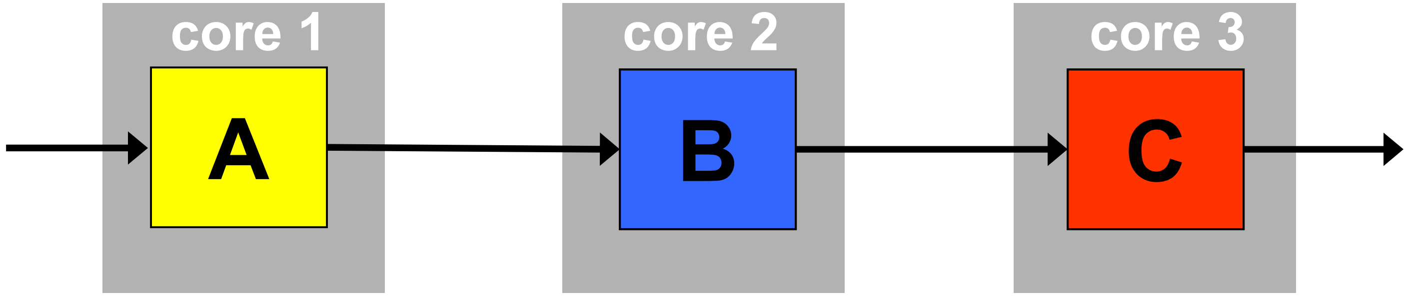
# CFAR Benchmark

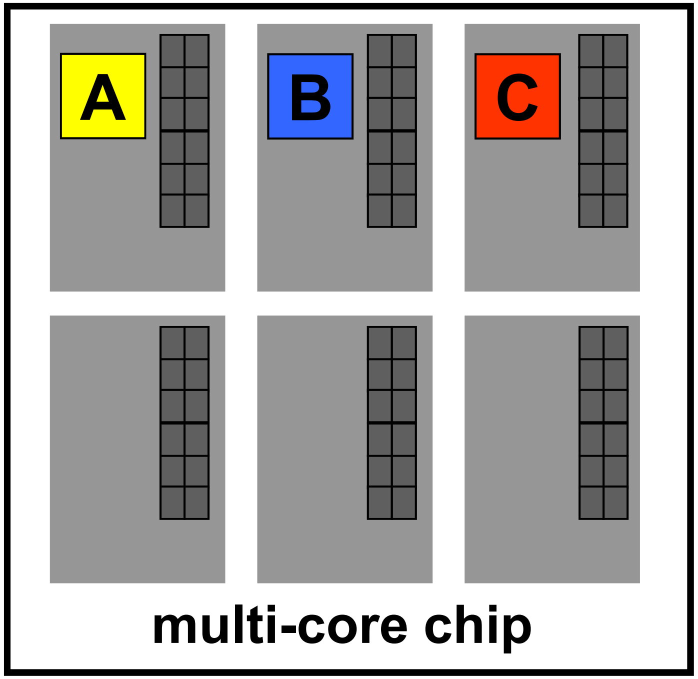


# Mapping



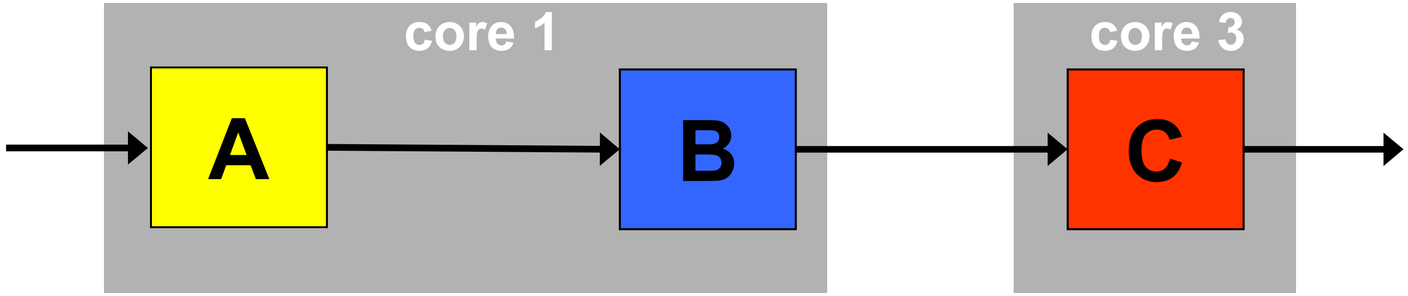
# Mapping

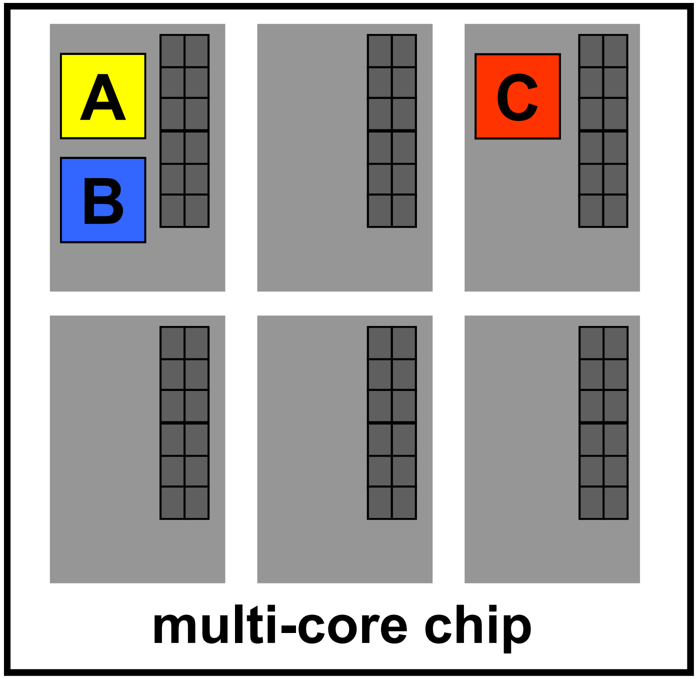



**Throughput**: rate of processing data tokens

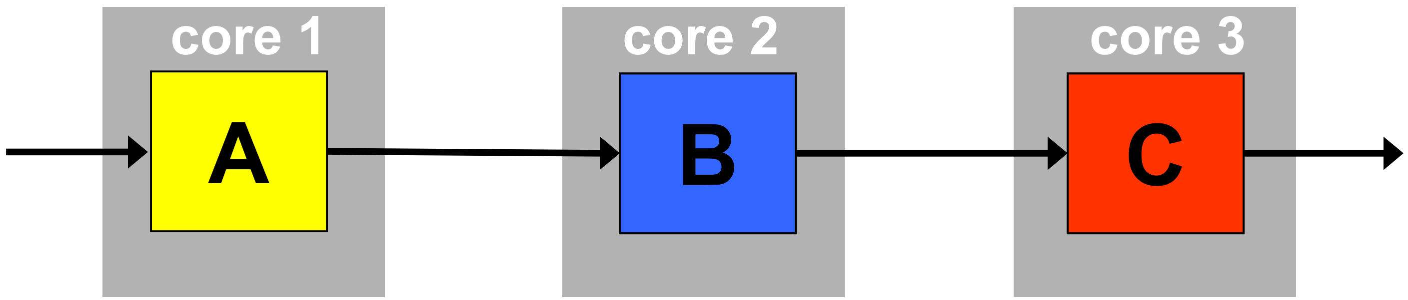
# Mapping



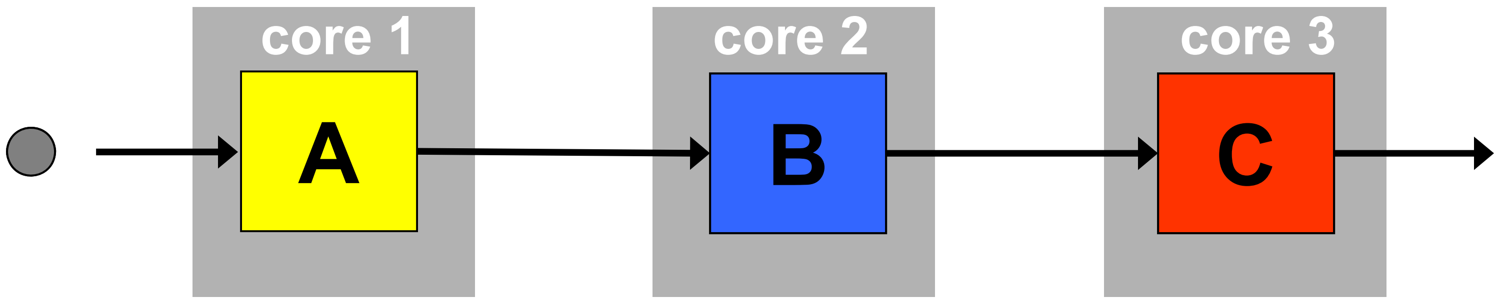

**Throughput**: rate of processing data tokens

# Unbalanced Flow



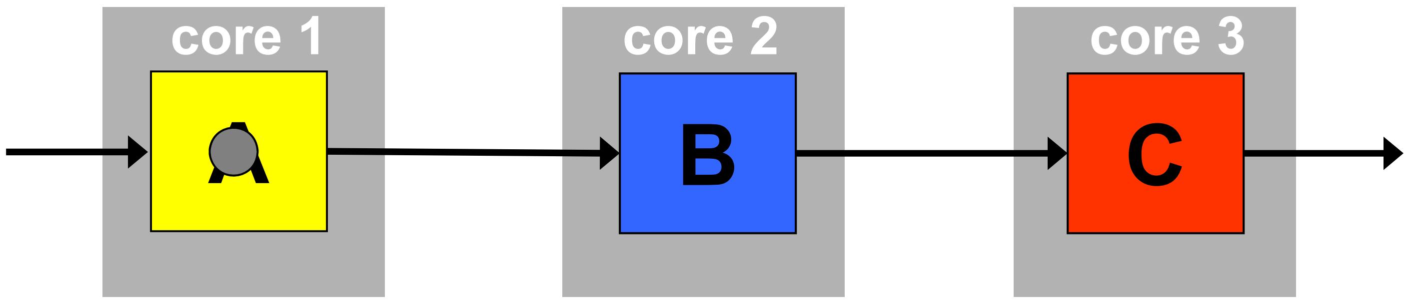

- Bottlenecks reduce throughput
  - Caused by backpressure
    - inherent algorithmic imbalances
    - data-dependent computational spikes

# Unbalanced Flow



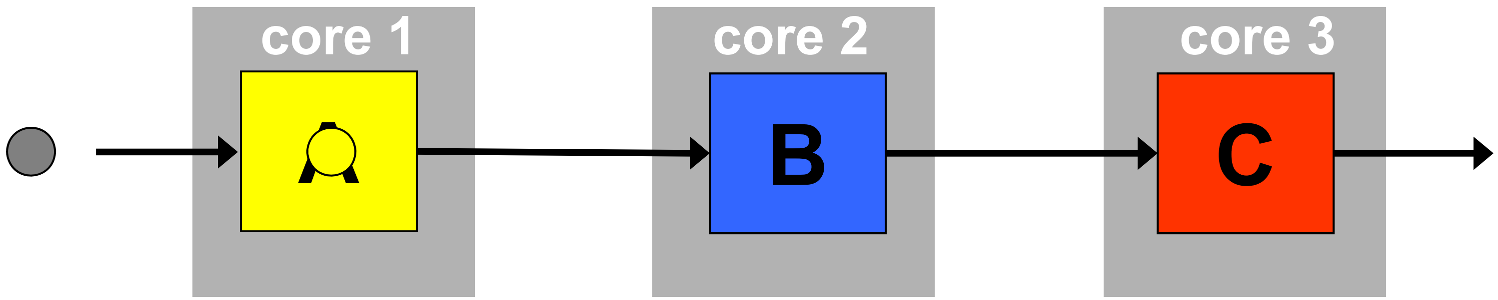

- **Bottlenecks reduce throughput**
  - **Caused by backpressure**
    - **inherent algorithmic imbalances**
    - **data-dependent computational spikes**

# Unbalanced Flow



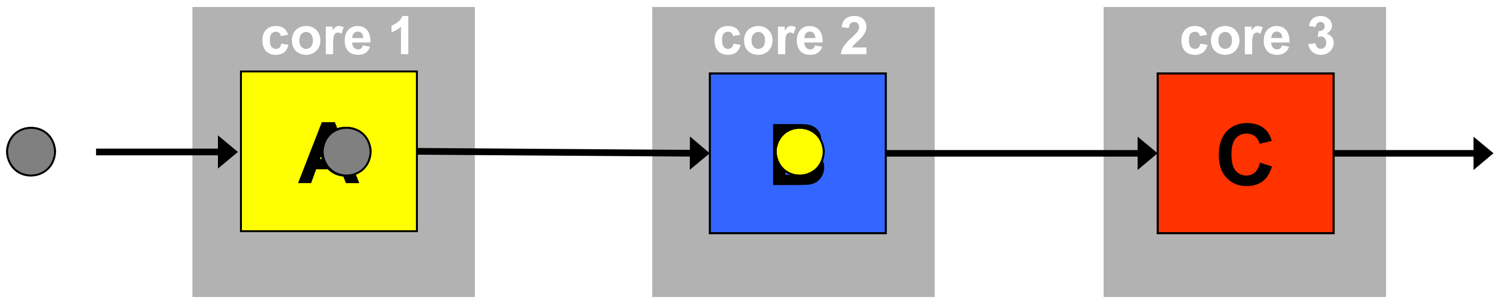

- Bottlenecks reduce throughput
  - Caused by backpressure
    - inherent algorithmic imbalances
    - data-dependent computational spikes

# Unbalanced Flow



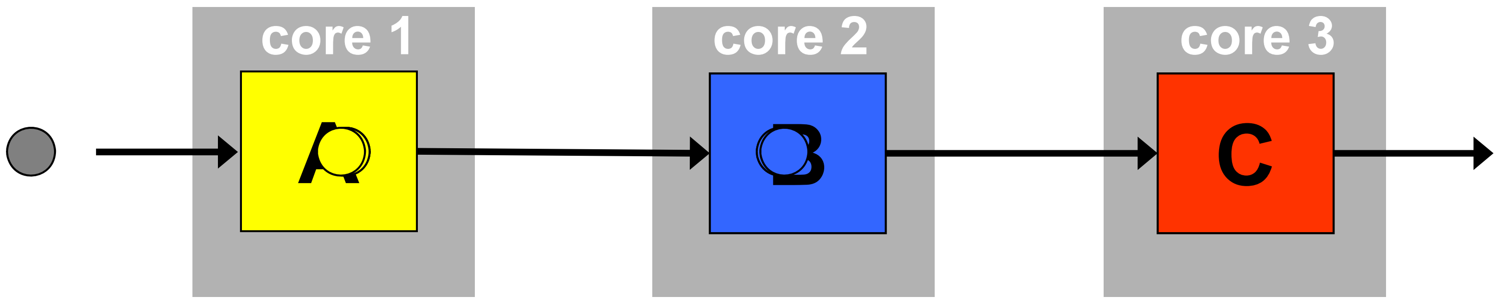

- **Bottlenecks reduce throughput**
  - **Caused by backpressure**
    - **inherent algorithmic imbalances**
    - **data-dependent computational spikes**

# Unbalanced Flow



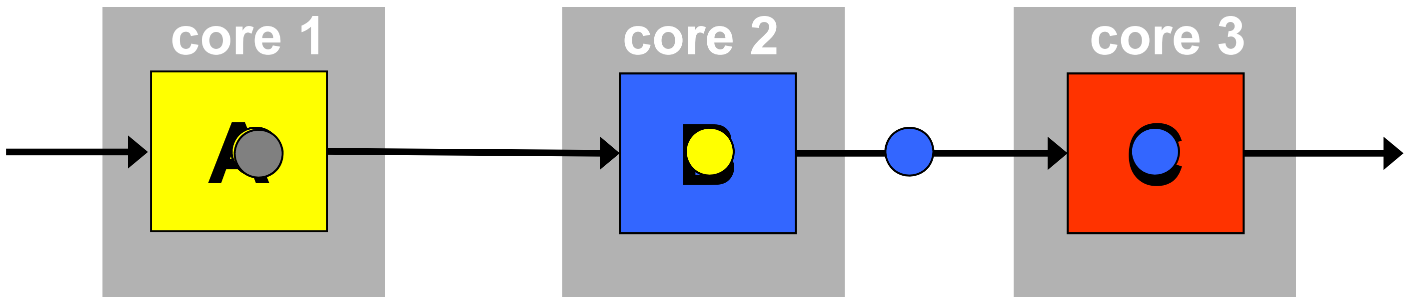

- Bottlenecks reduce throughput
  - Caused by backpressure
    - inherent algorithmic imbalances
    - data-dependent computational spikes

# Unbalanced Flow



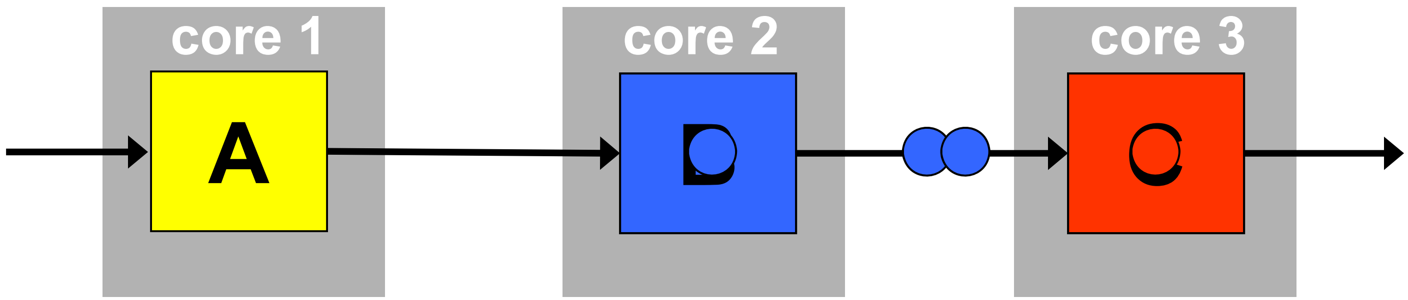

- Bottlenecks reduce throughput
  - Caused by backpressure
    - inherent algorithmic imbalances
    - data-dependent computational spikes

# Unbalanced Flow



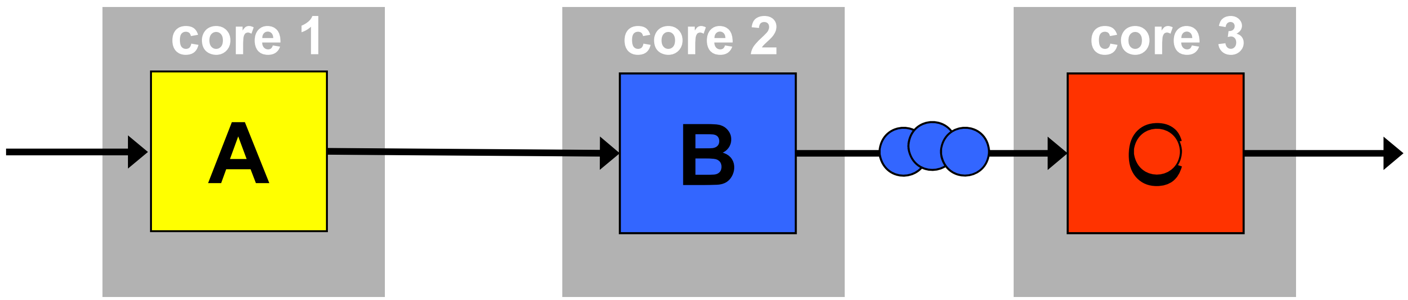

- **Bottlenecks reduce throughput**
  - **Caused by backpressure**
    - **inherent algorithmic imbalances**
    - **data-dependent computational spikes**

# Unbalanced Flow



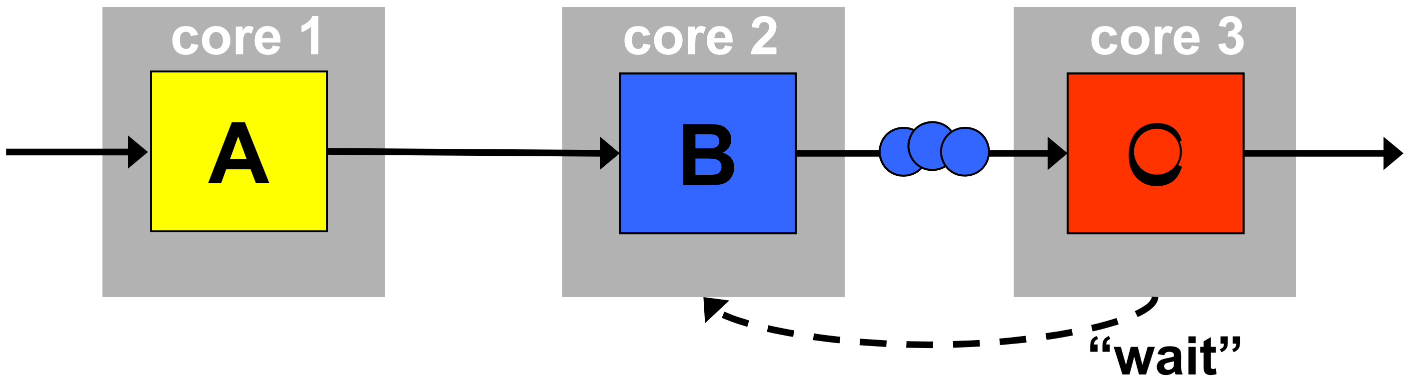

- **Bottlenecks reduce throughput**
  - **Caused by backpressure**
    - **inherent algorithmic imbalances**
    - **data-dependent computational spikes**

# Unbalanced Flow



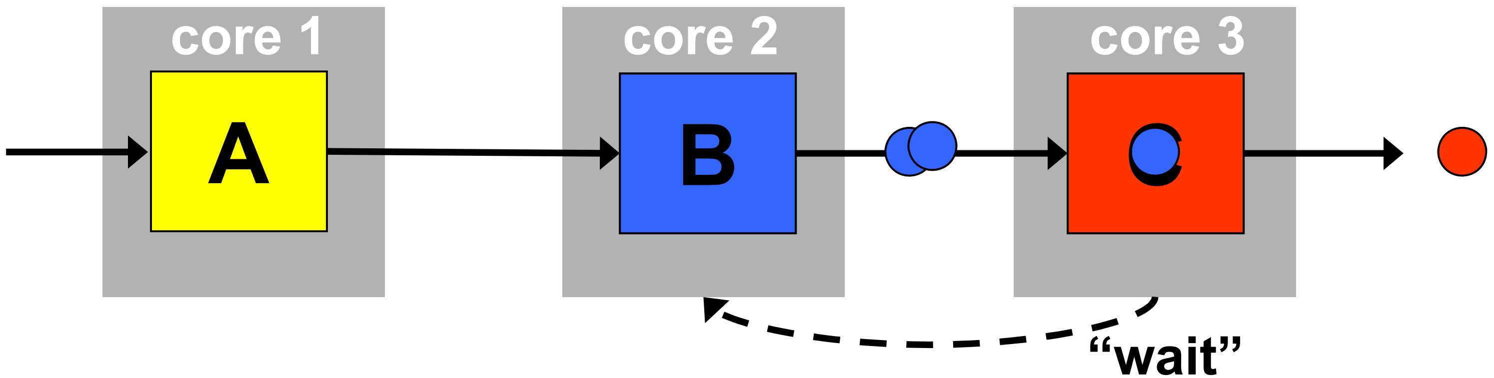

- Bottlenecks reduce throughput
  - Caused by backpressure
    - inherent algorithmic imbalances
    - data-dependent computational spikes

# Unbalanced Flow




- **Bottlenecks reduce throughput**
  - **Caused by backpressure**
    - **inherent algorithmic imbalances**
    - **data-dependent computational spikes**

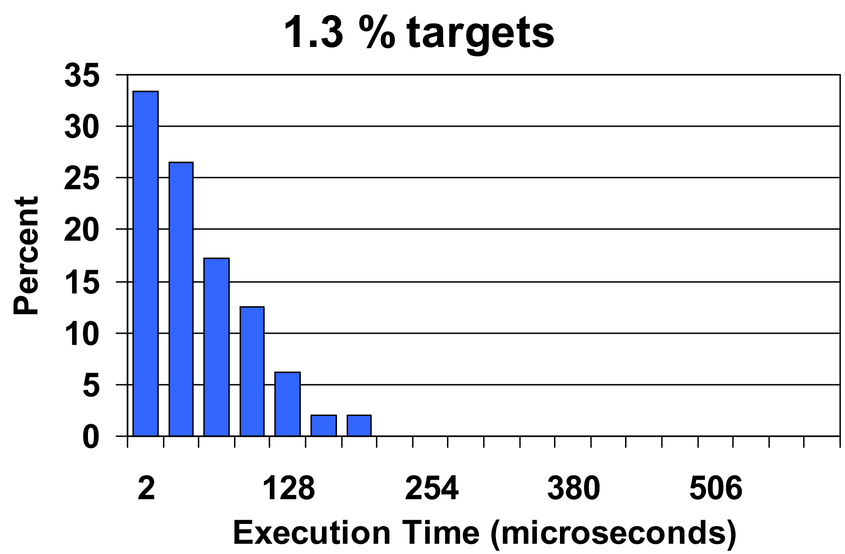
# Unbalanced Flow

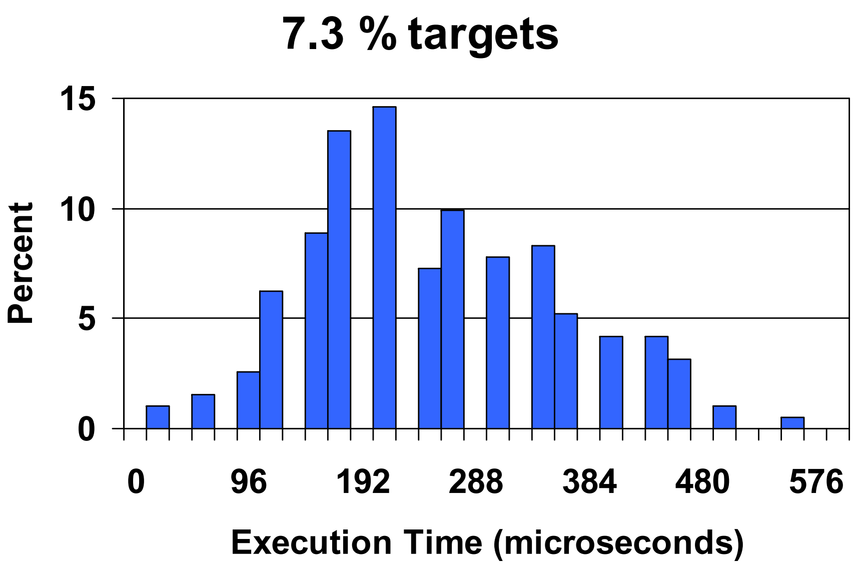


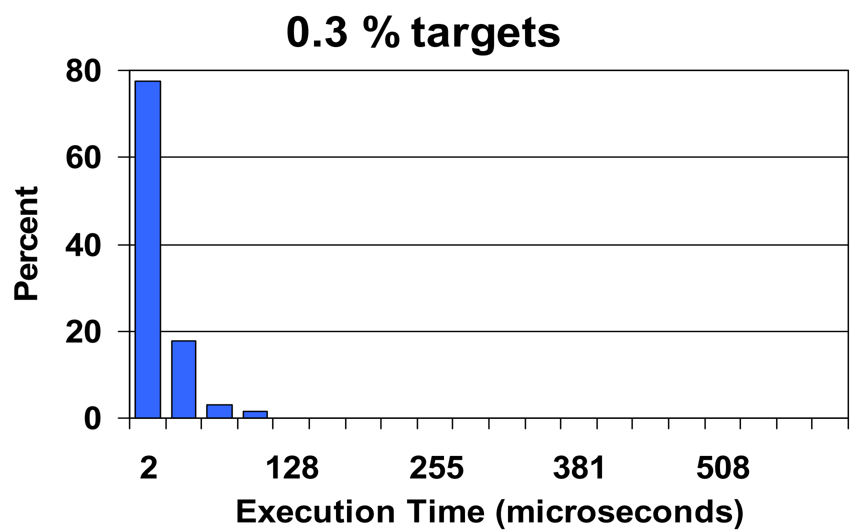

- Bottlenecks reduce throughput
  - Caused by backpressure
    - inherent algorithmic imbalances
    - data-dependent computational spikes

# Data Dependent Execution Time: CFAR

- Using Set 1 from the HPEC CFAR Kernel Benchmark
- Over a block of about 100 cells
- Extra workload of 32 microseconds per target



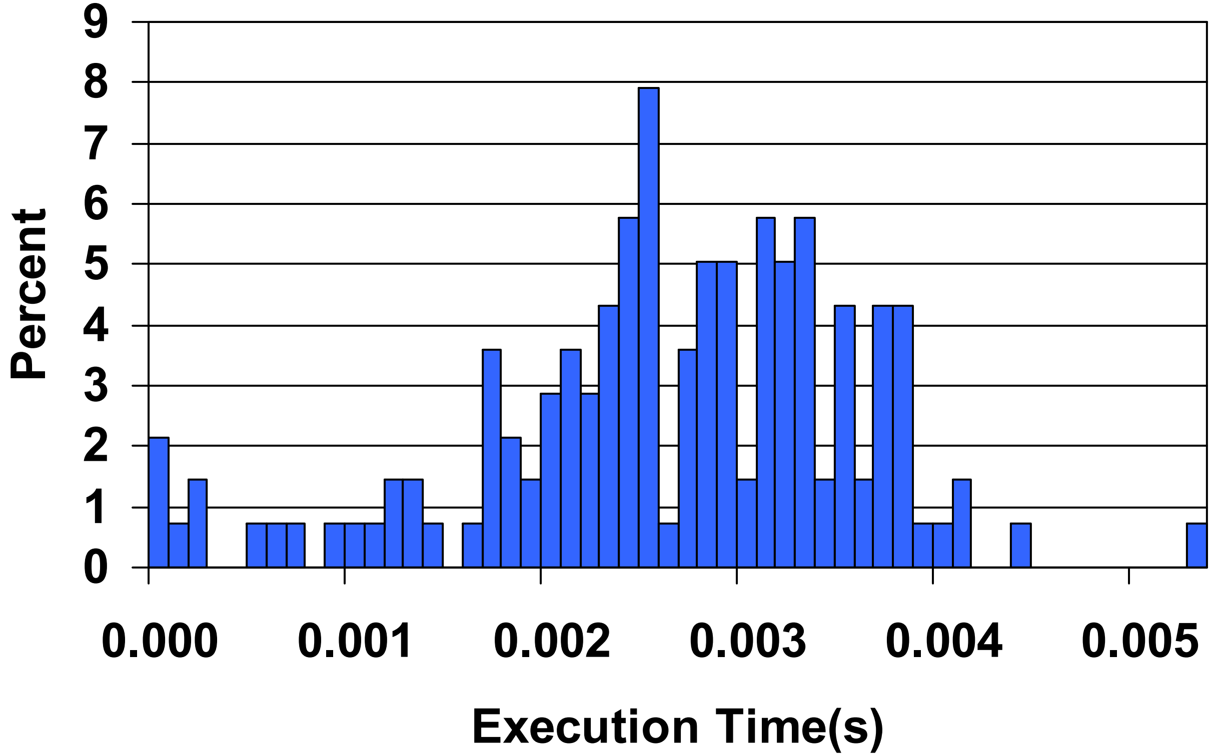



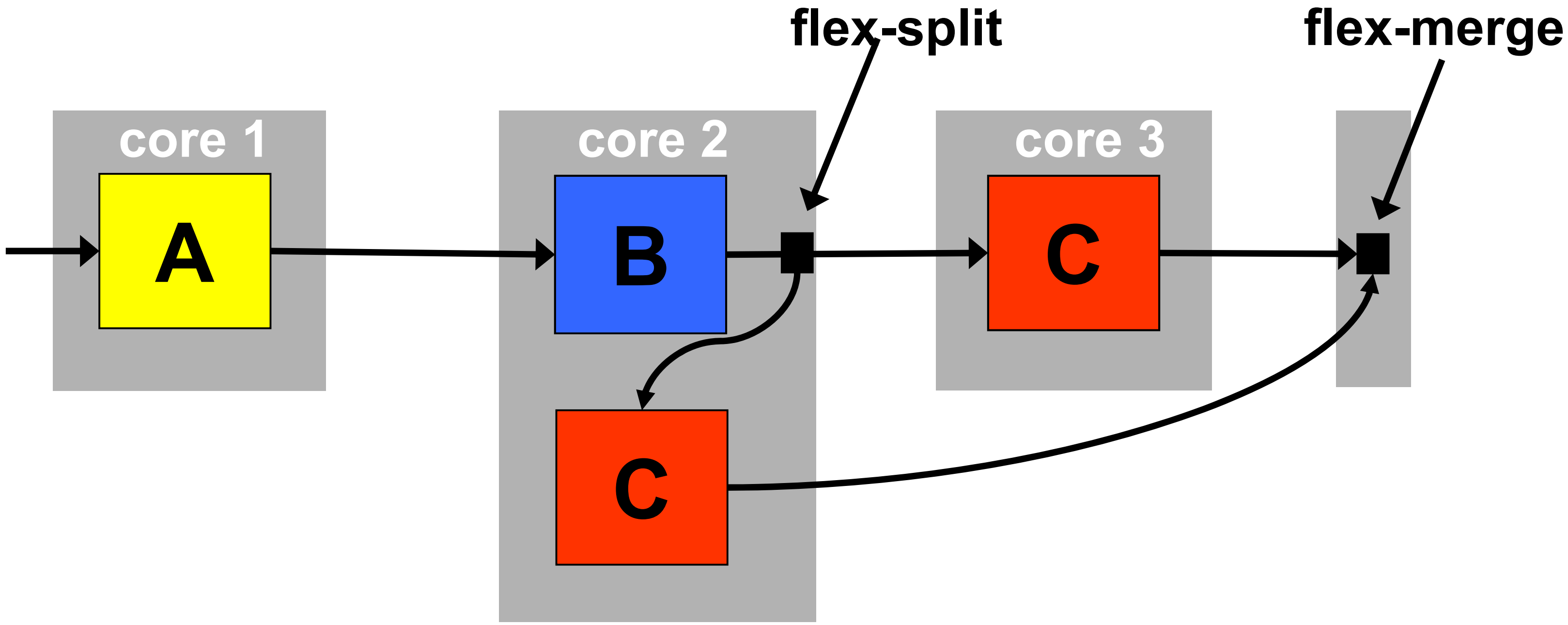
# Data Dependent Execution Time

Some other examples:

- Bloom Filters (Financial, Spam detection)
- Compression (Image processing)

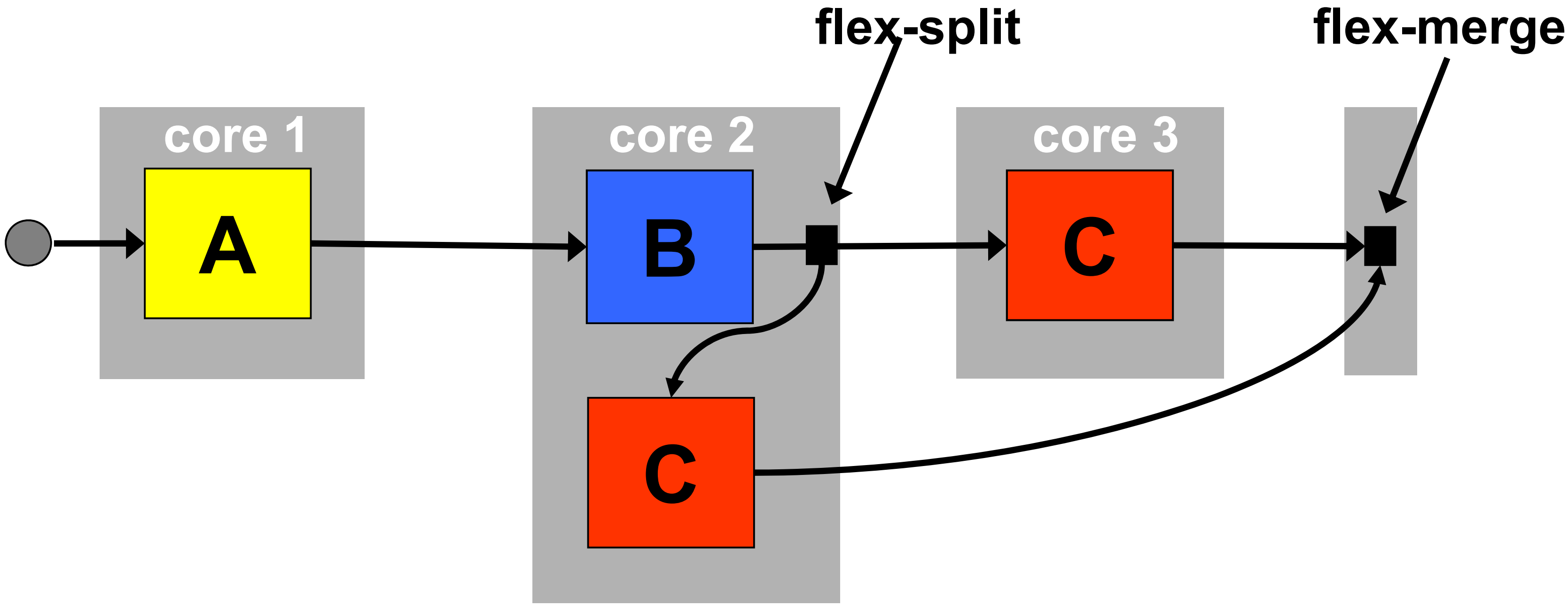


# Our Solution: Flexible Filters



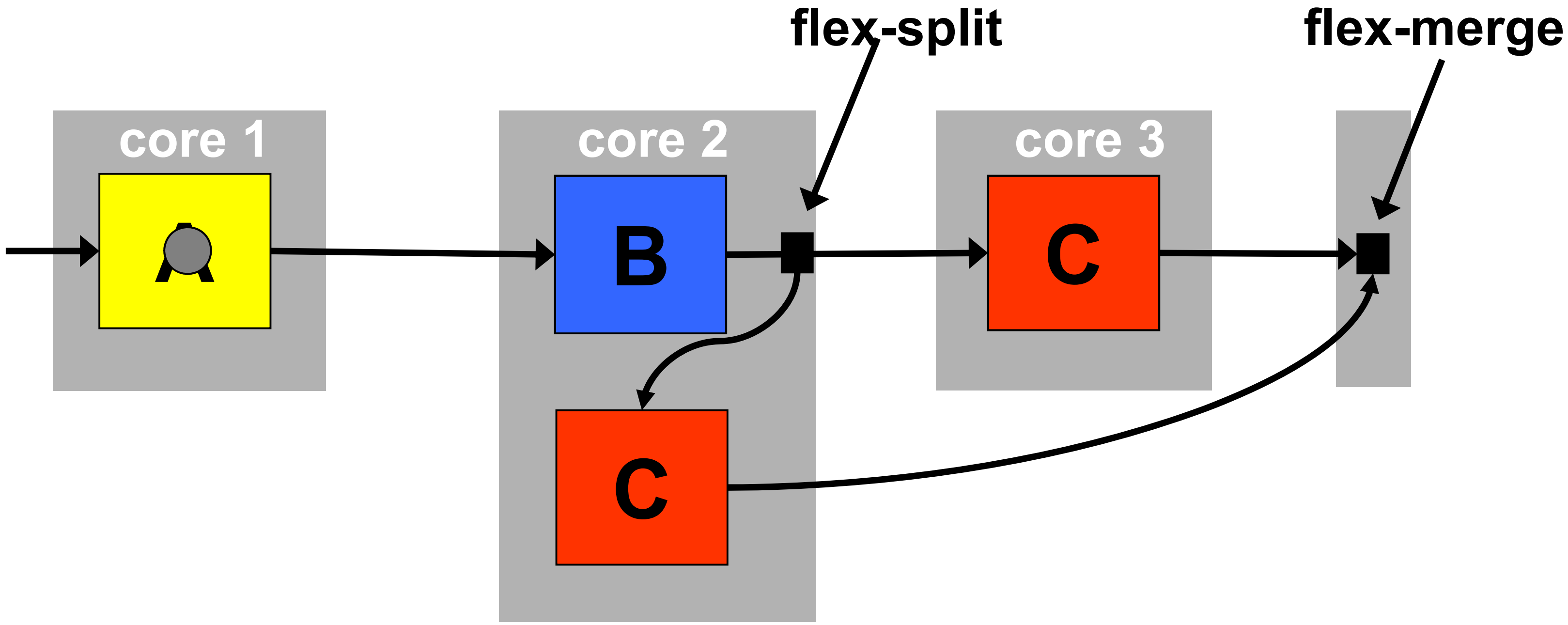

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



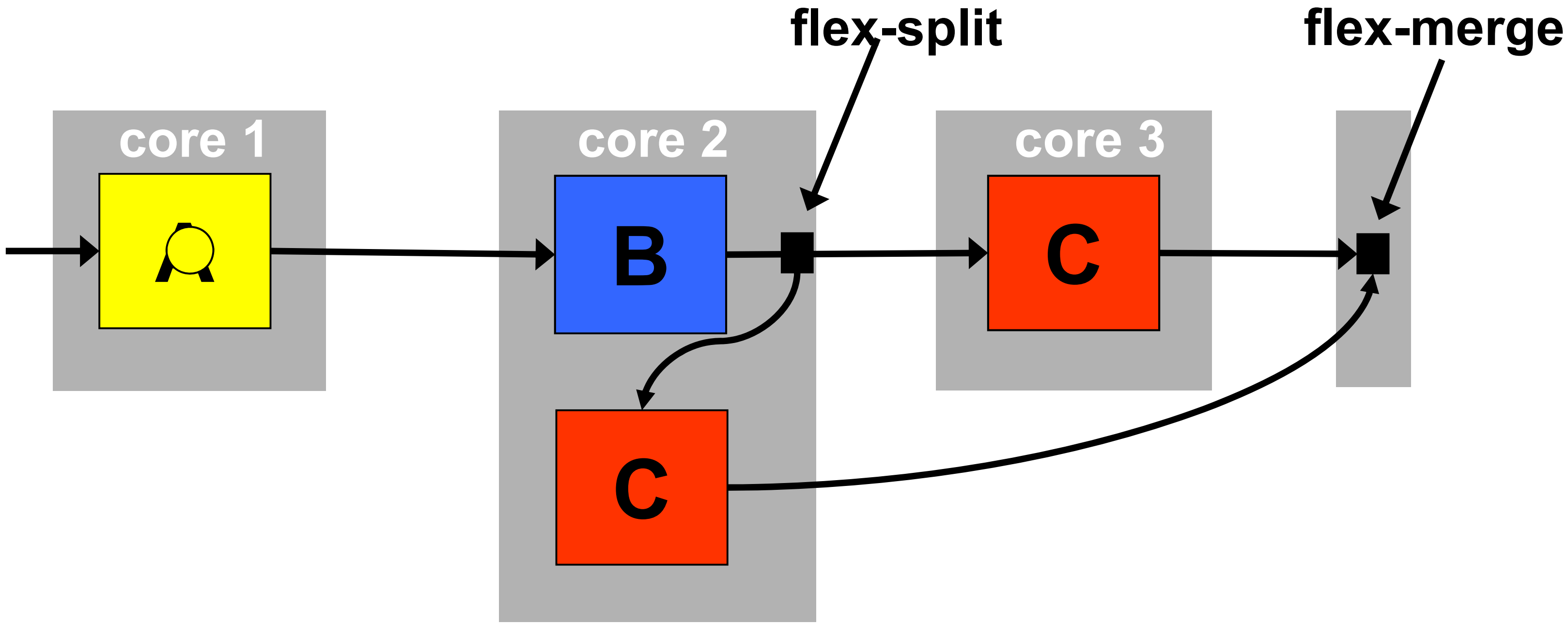

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



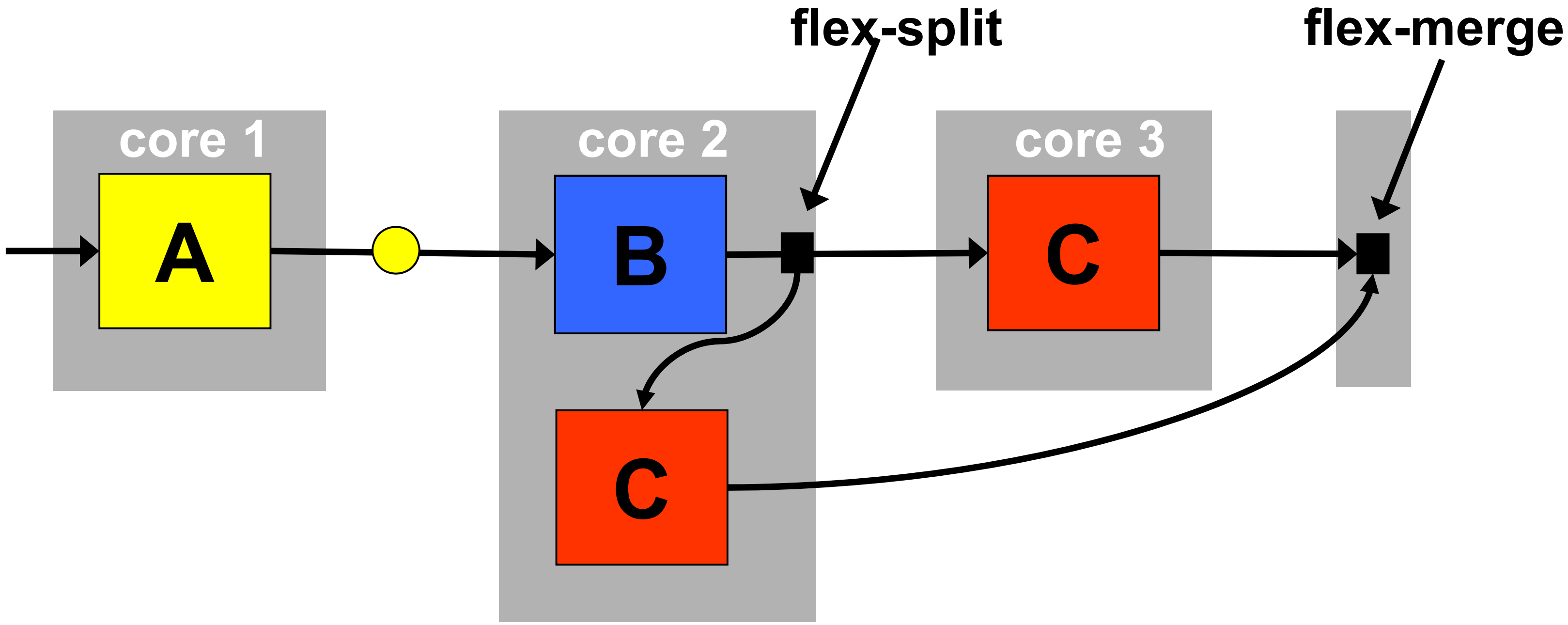

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



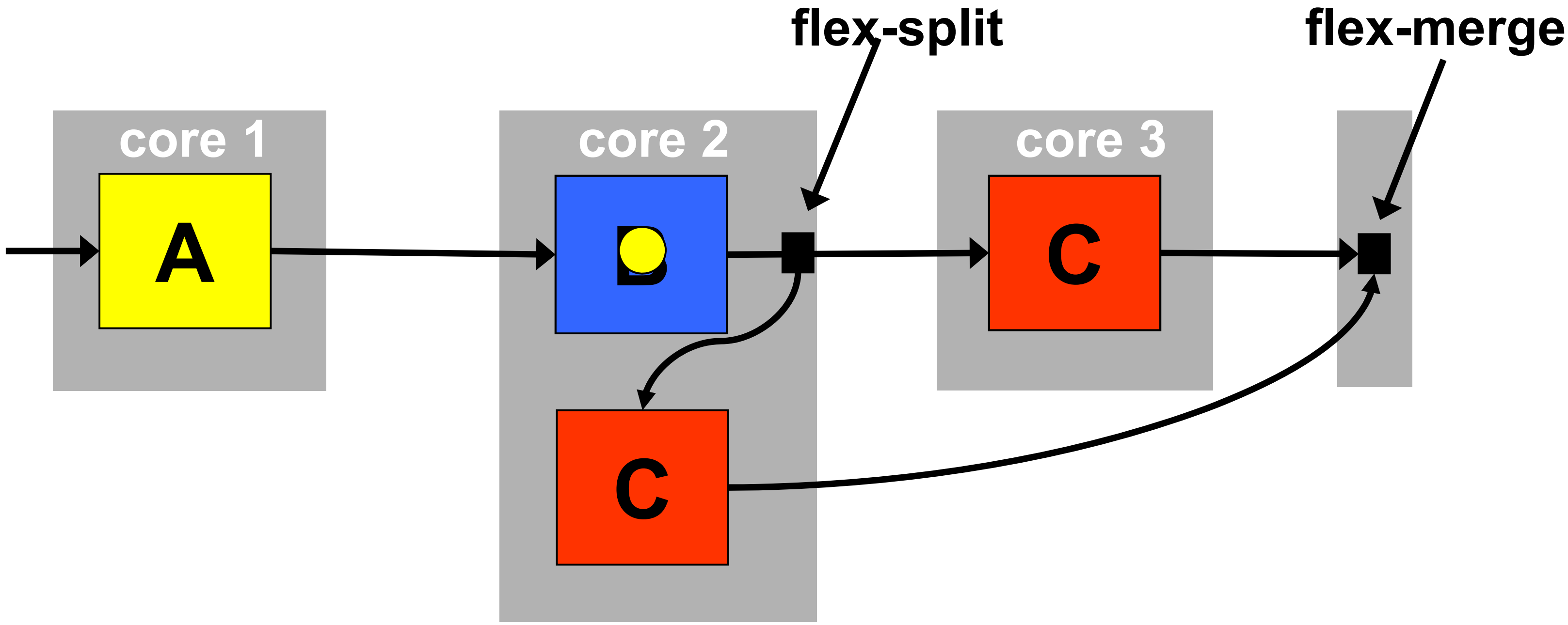

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



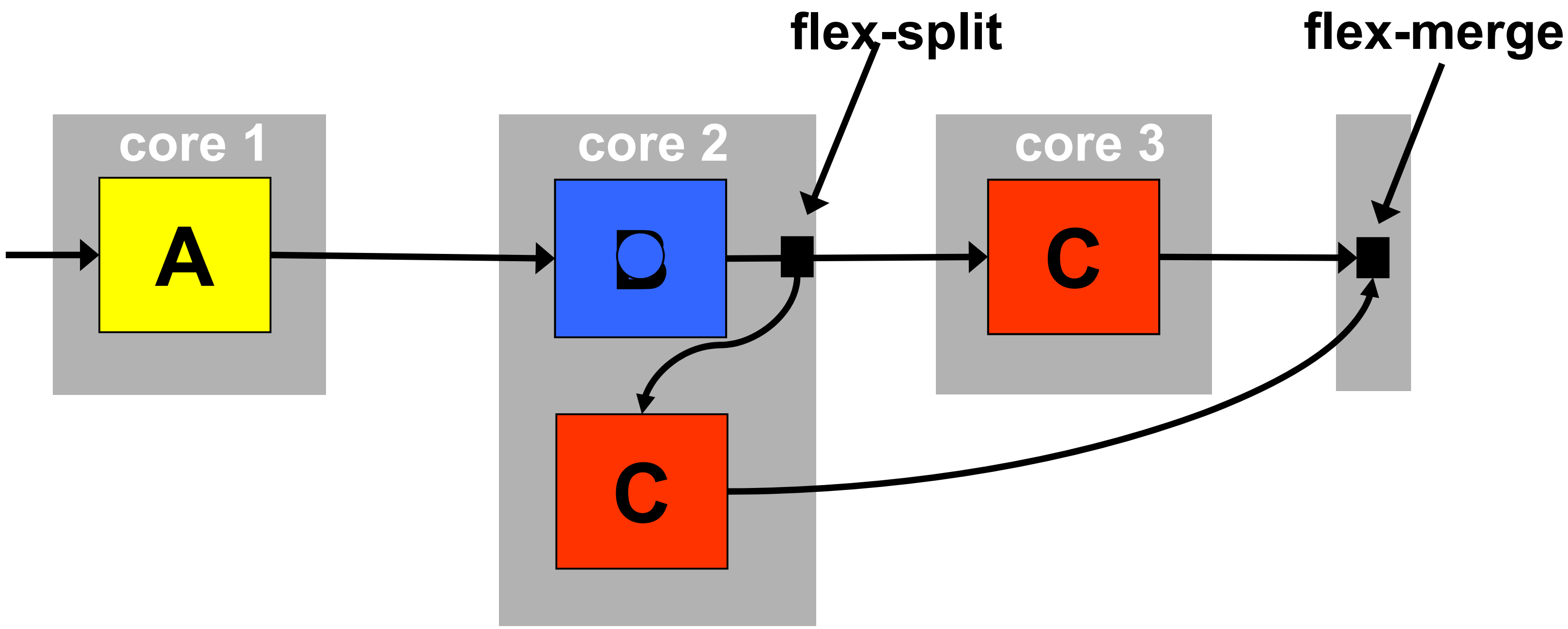

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



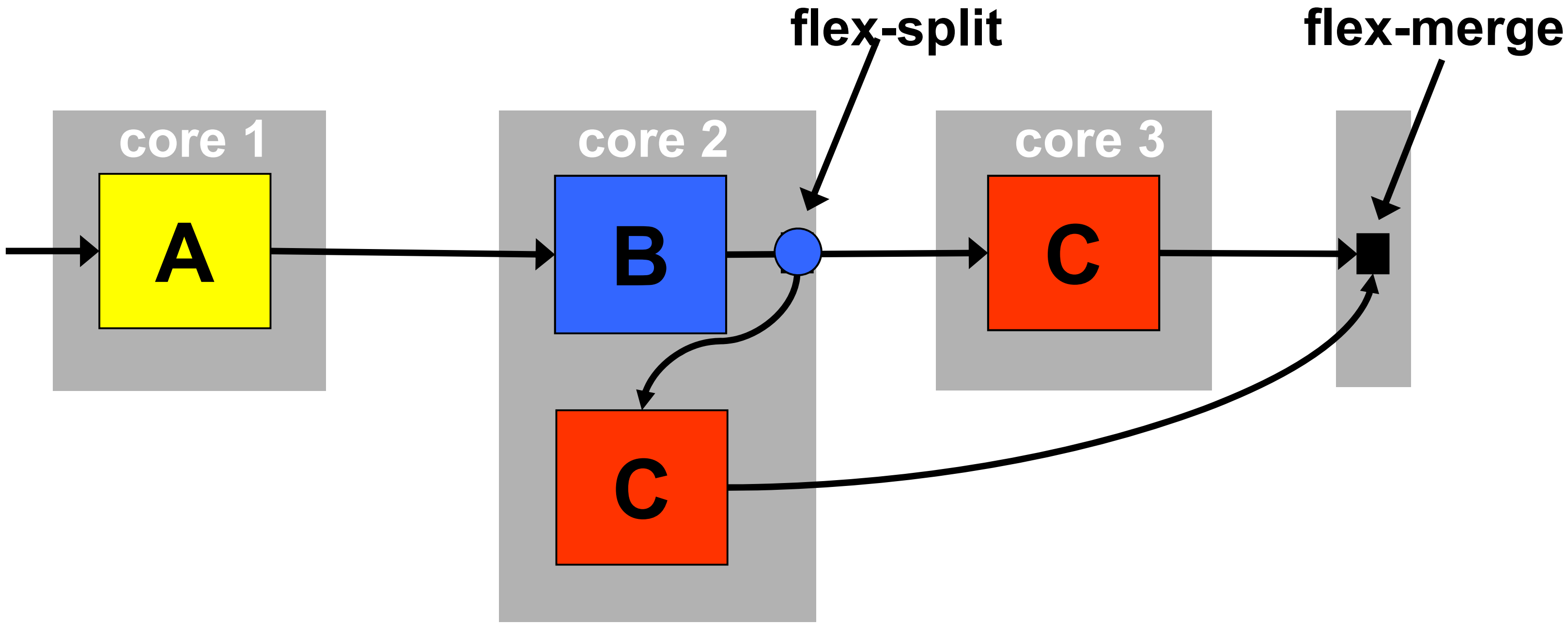

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



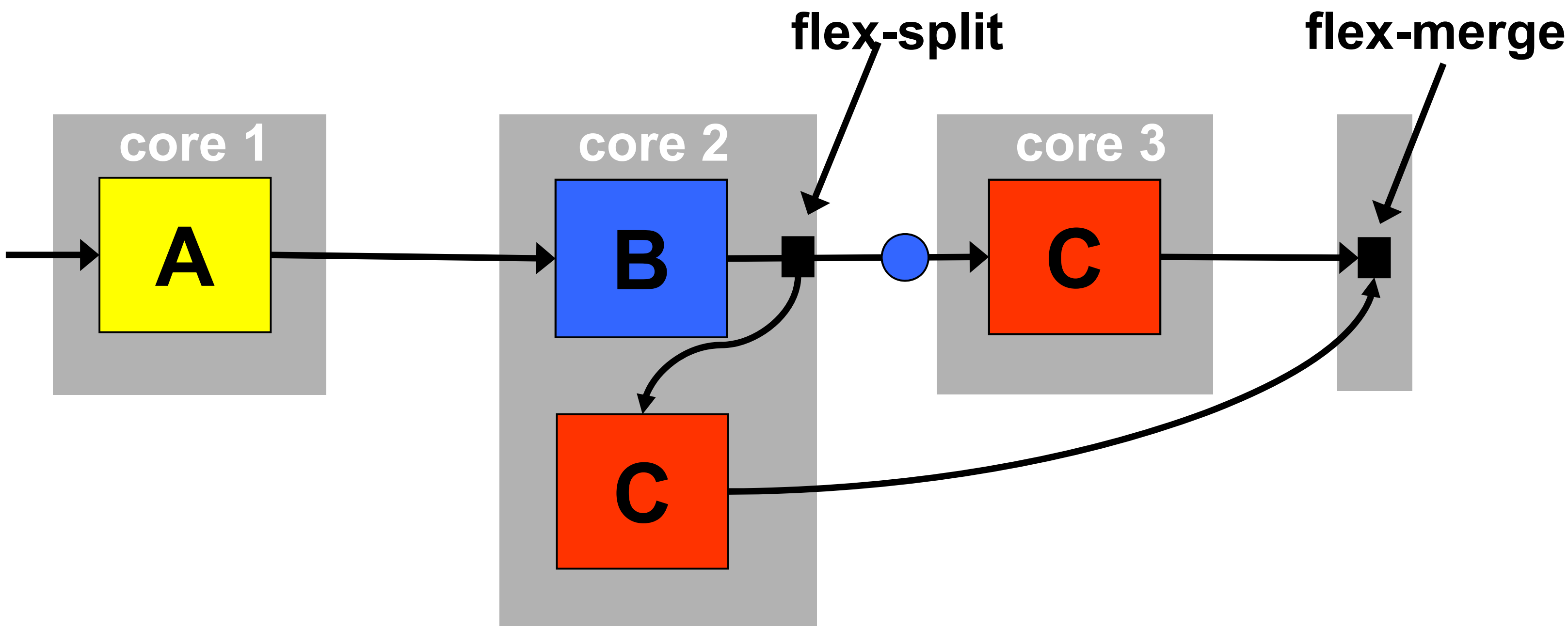

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



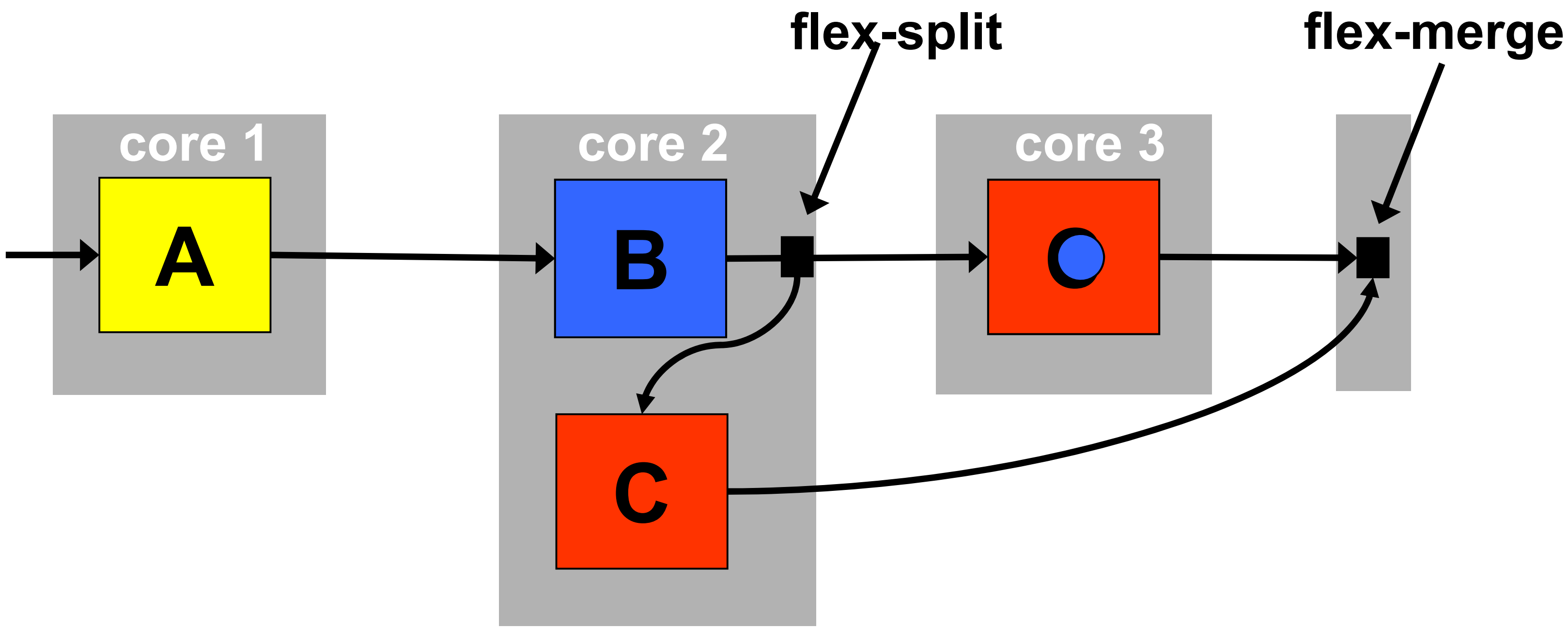

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



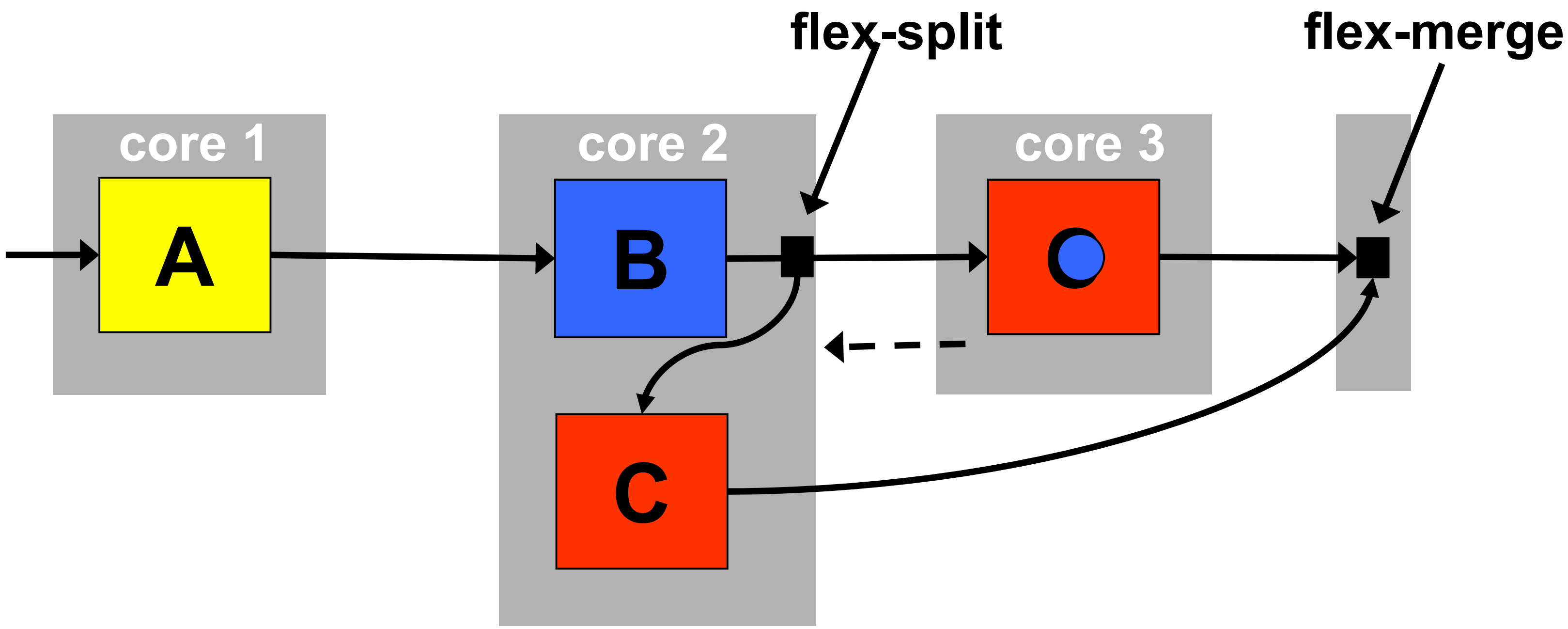

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



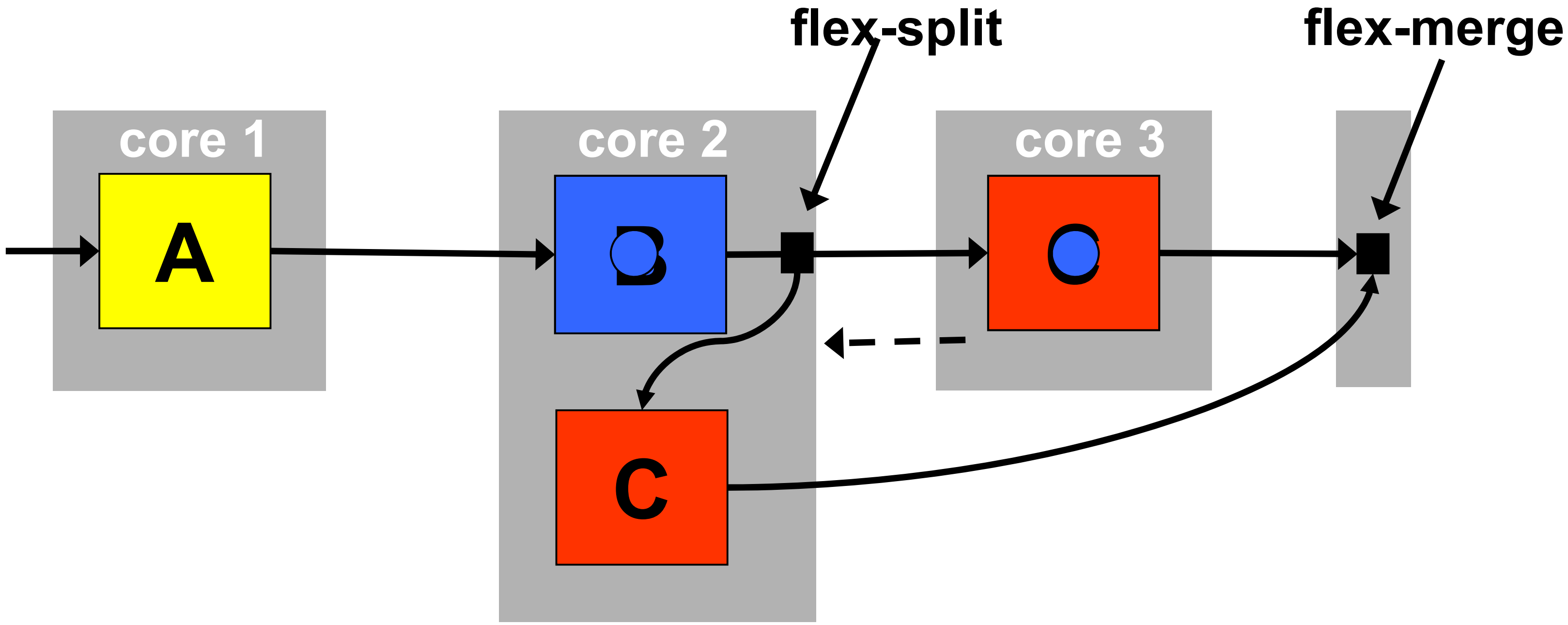

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



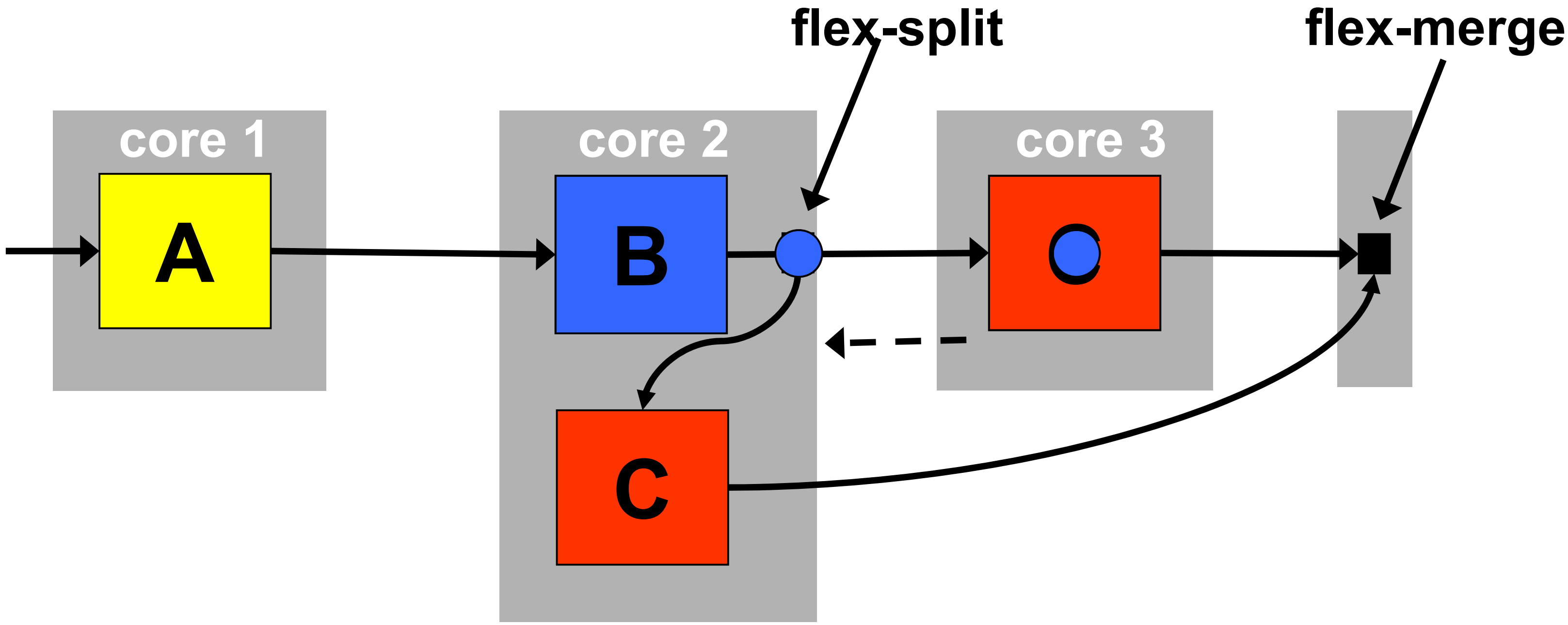

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



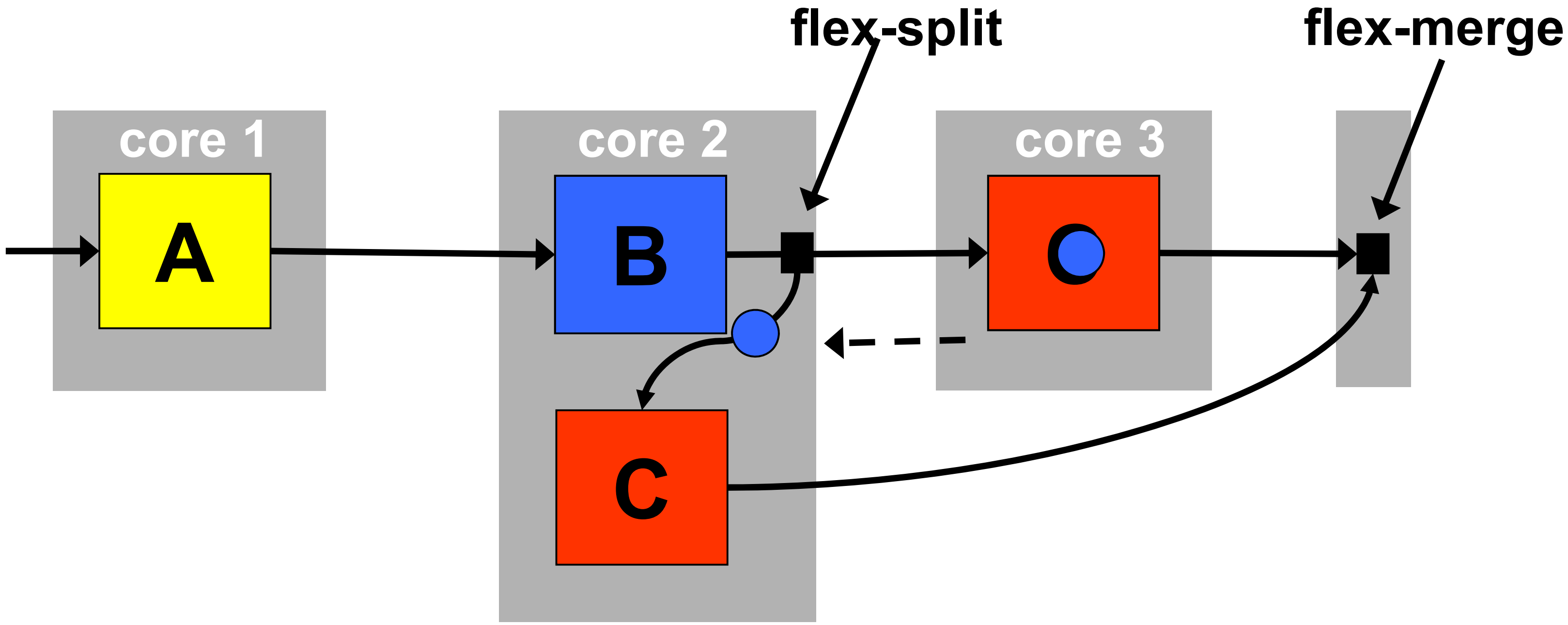

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



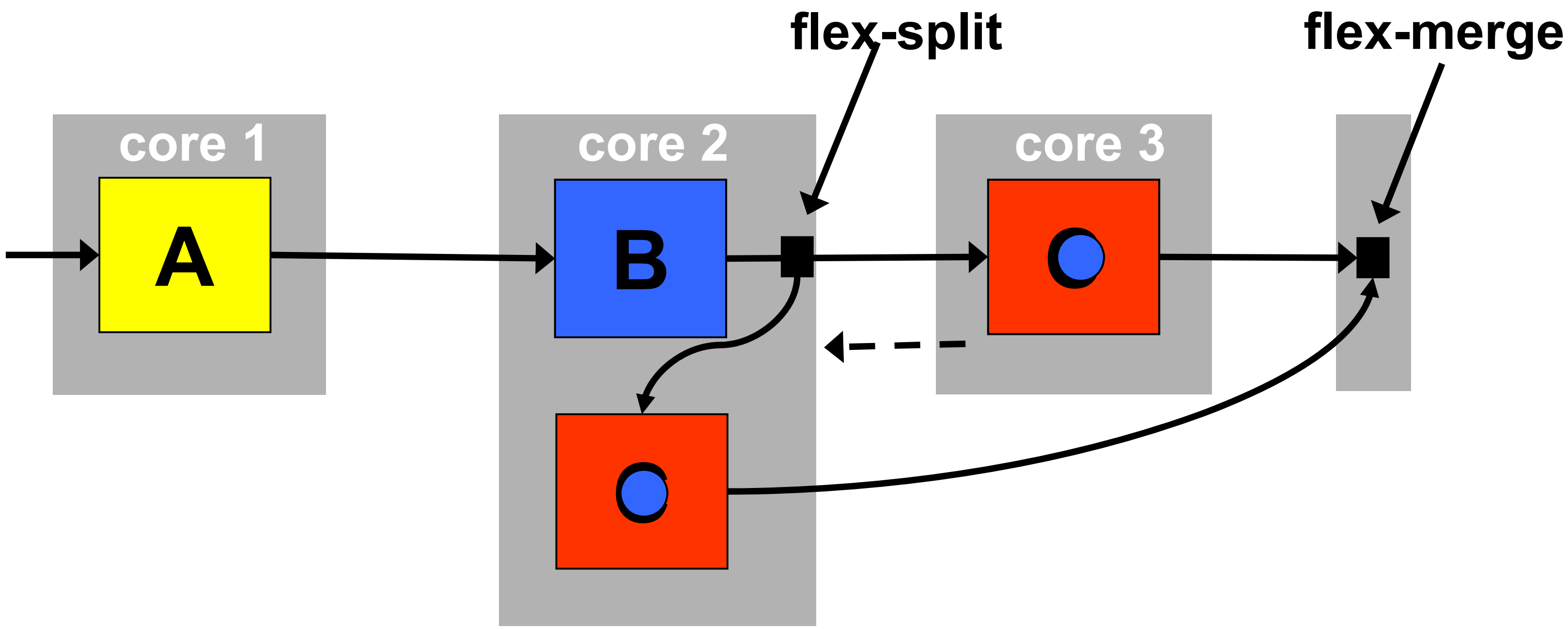

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



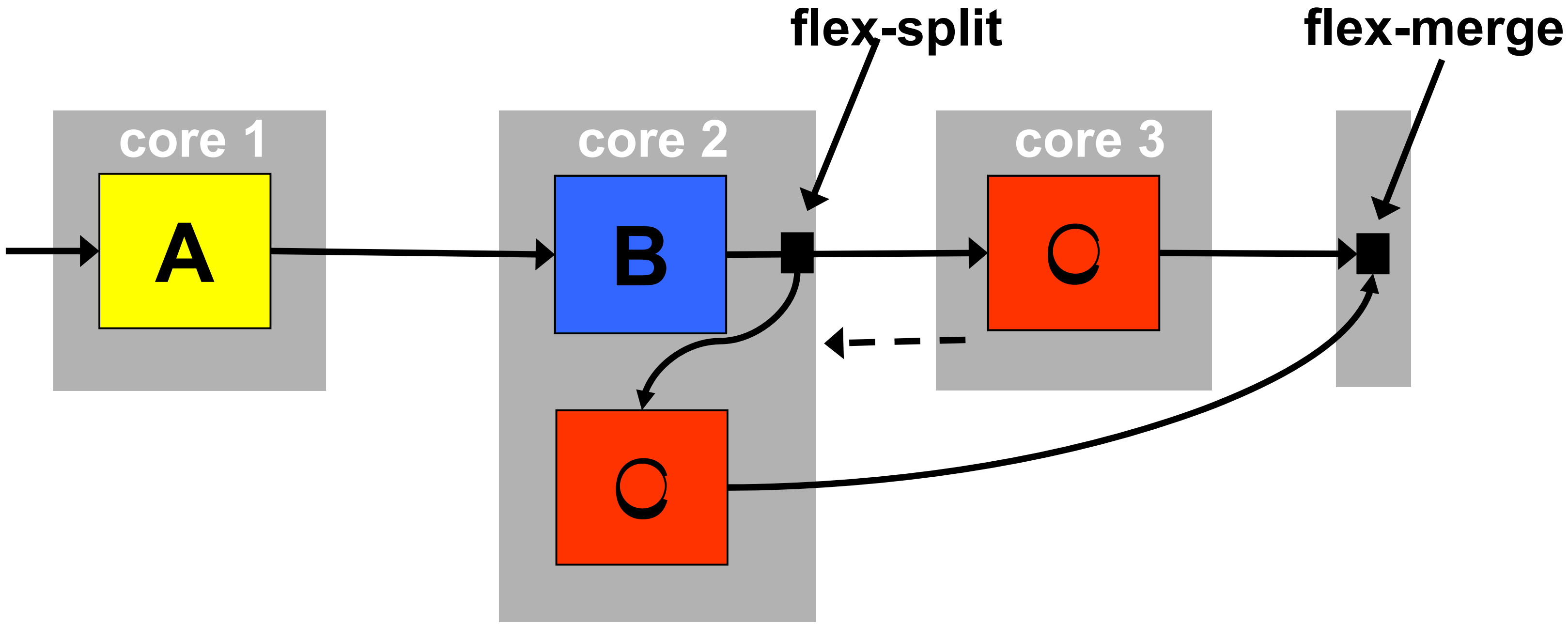

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



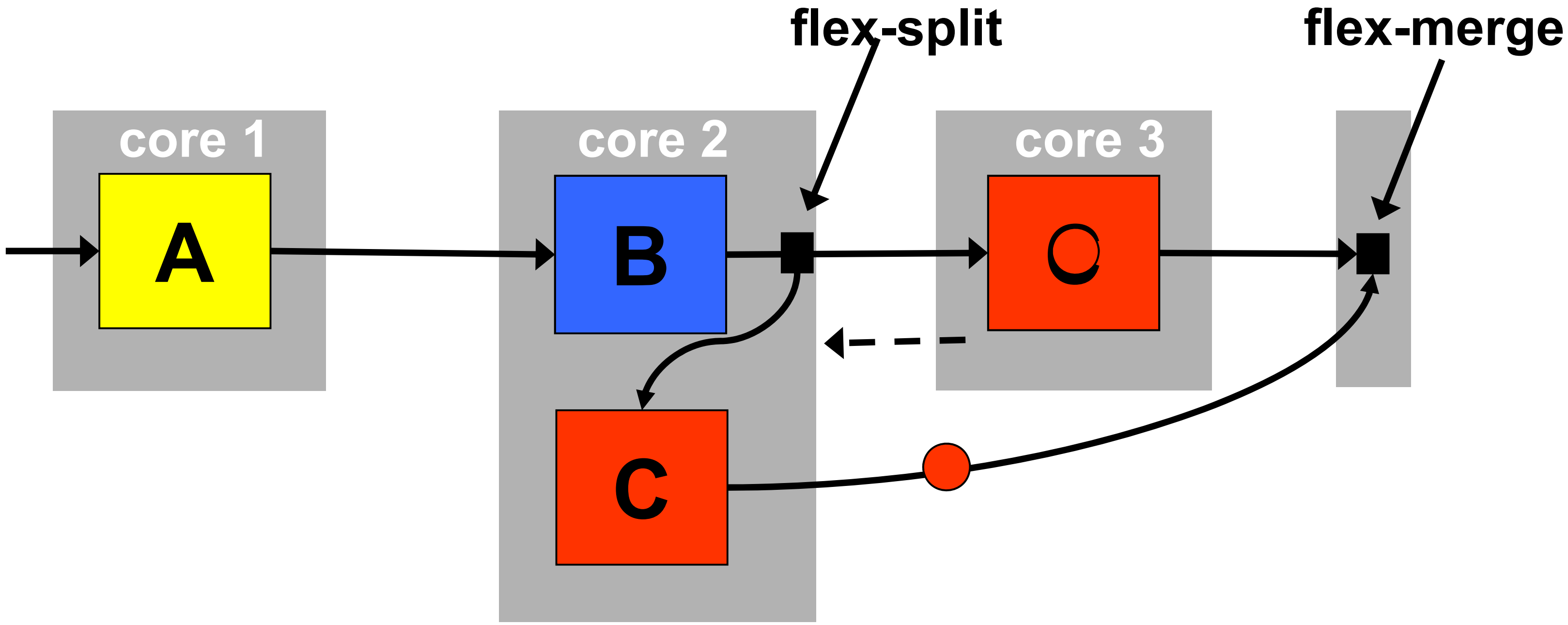

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



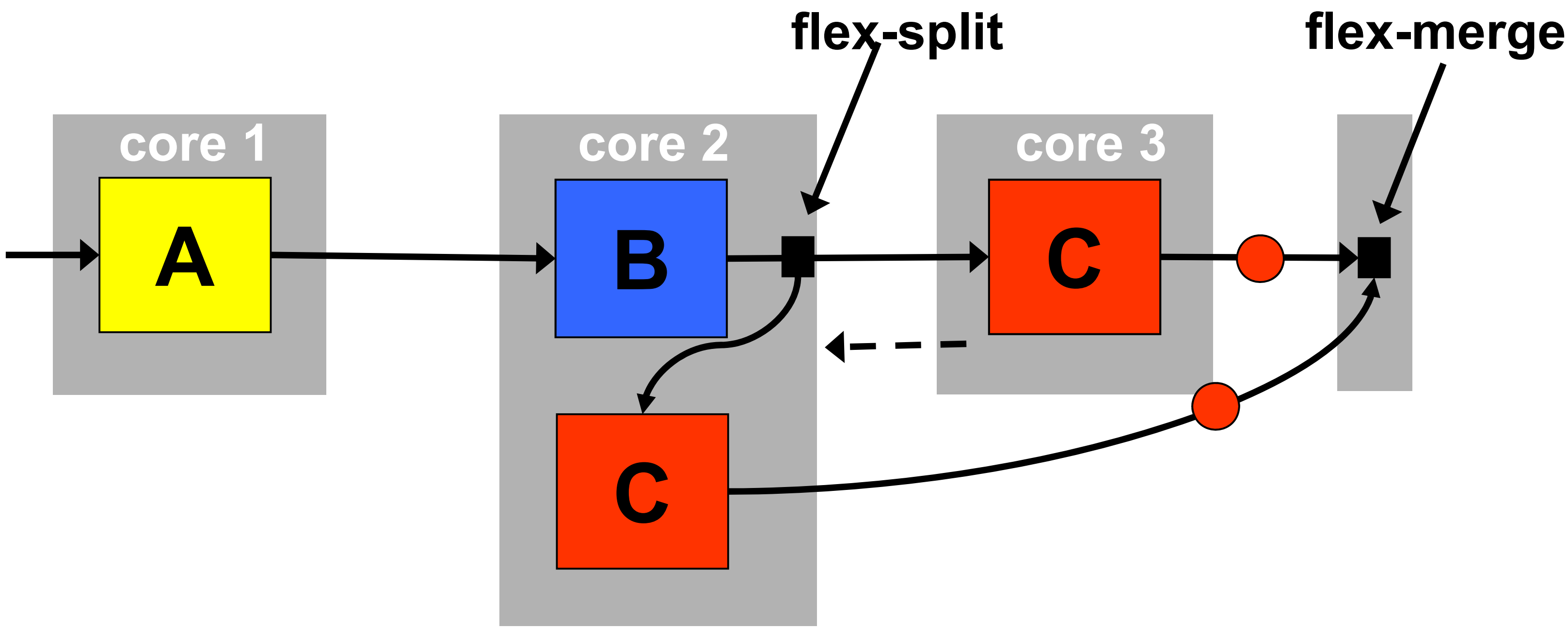

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



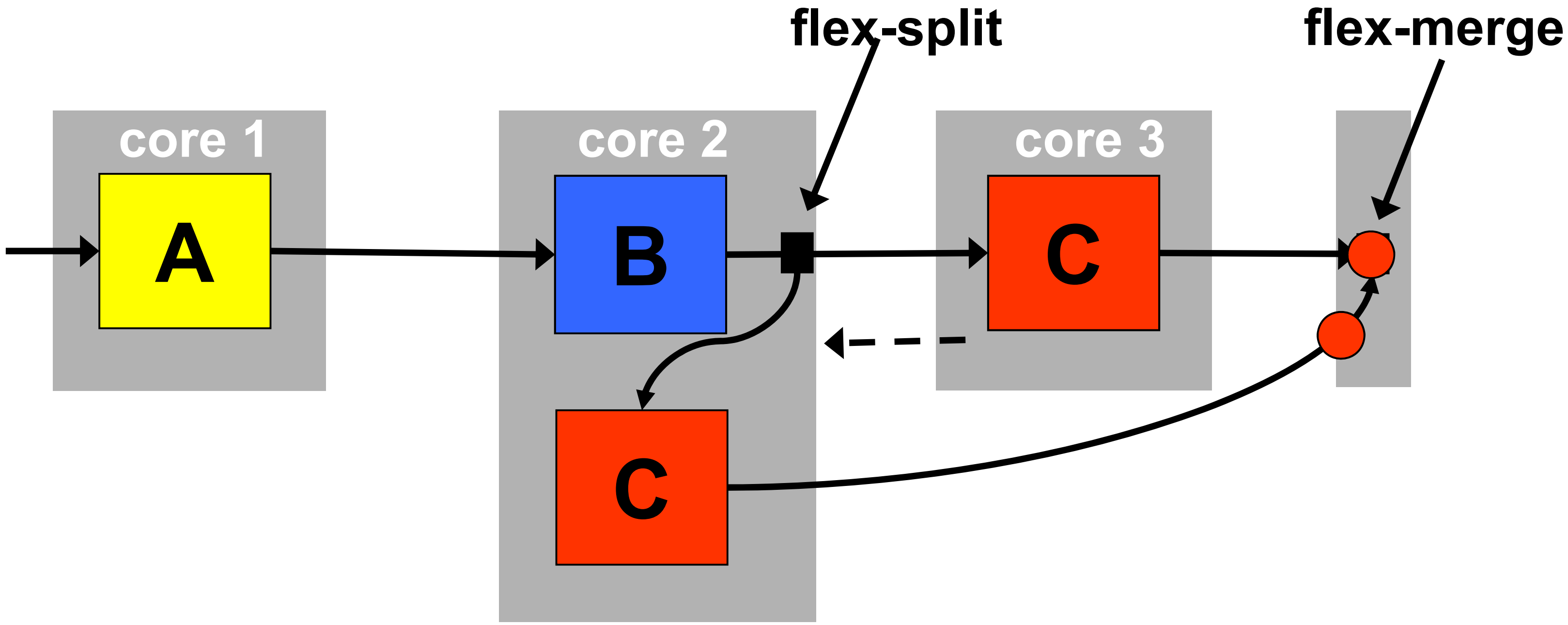

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



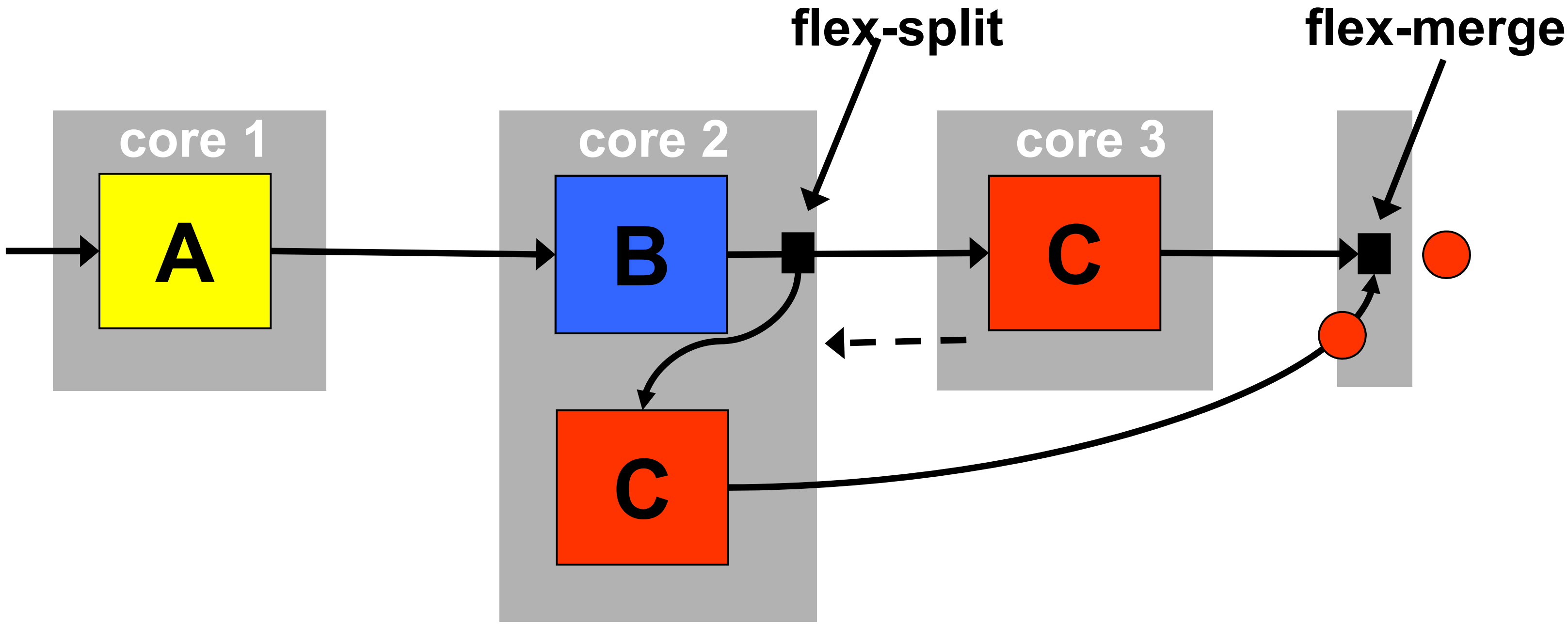

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



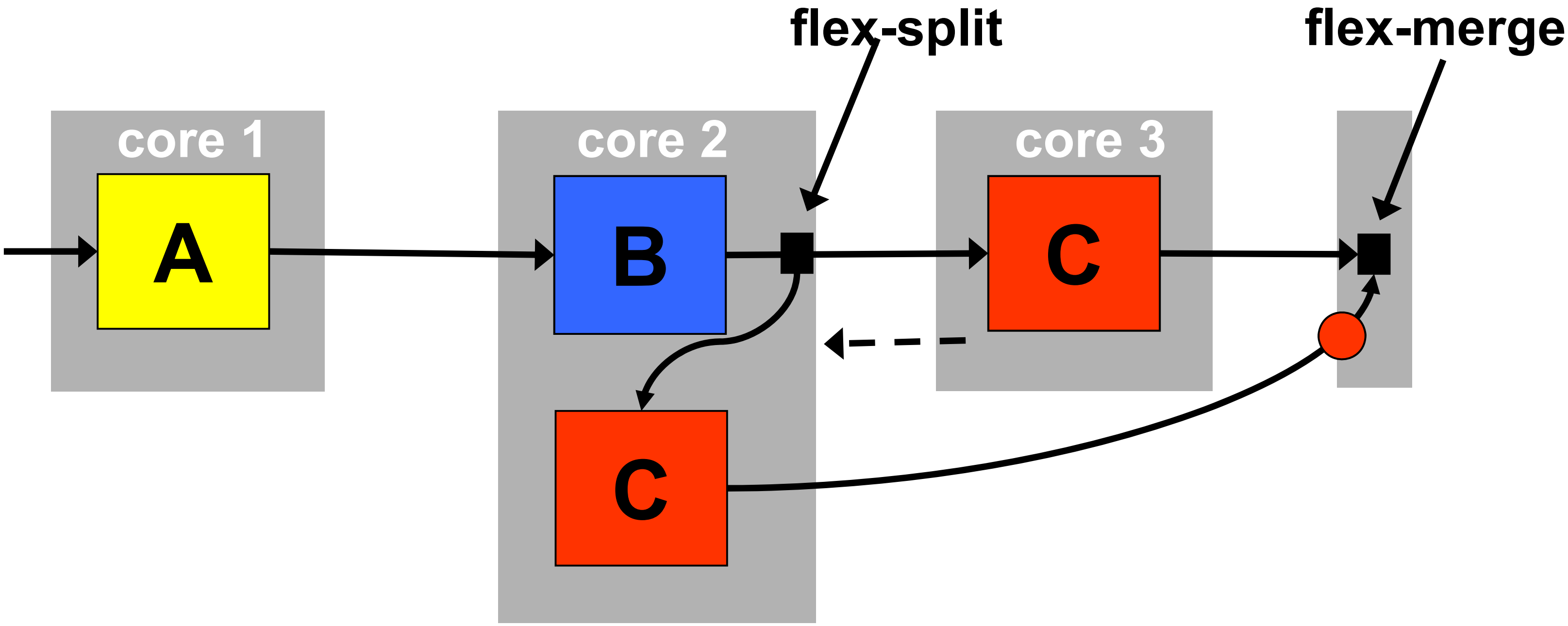

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



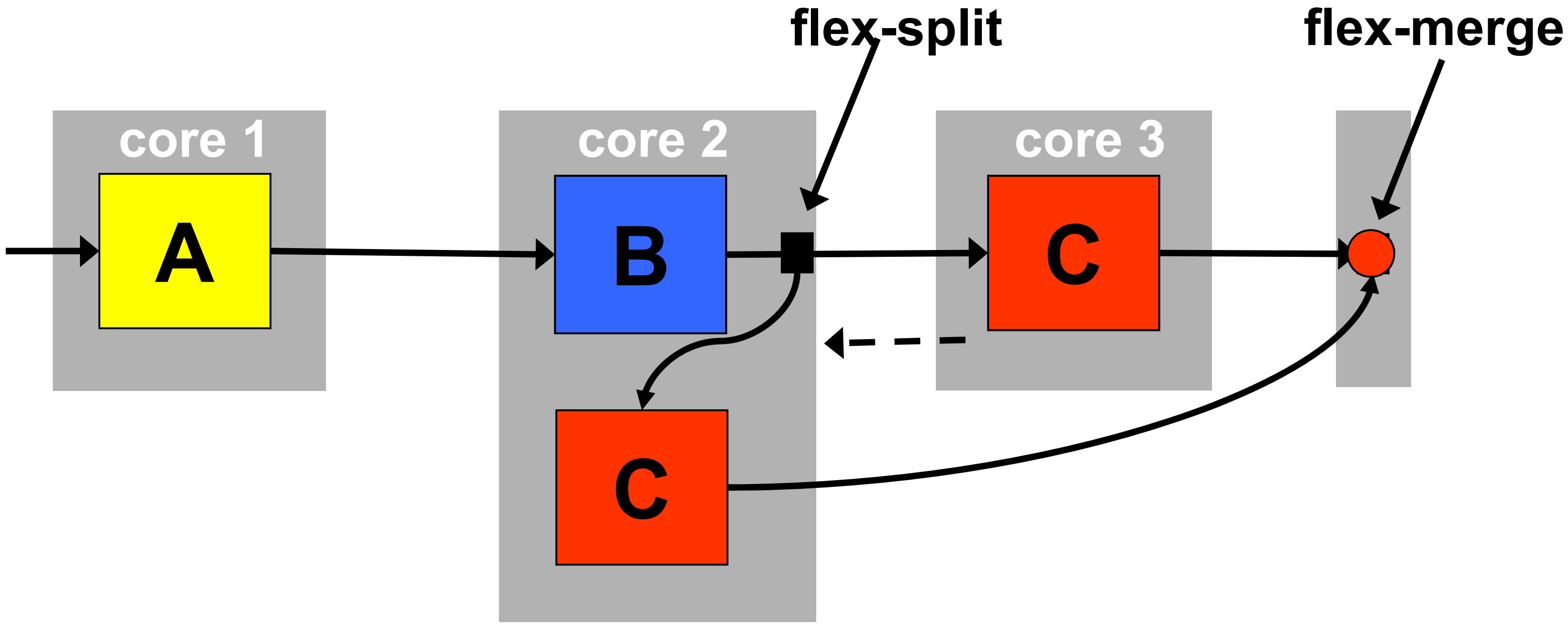

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters



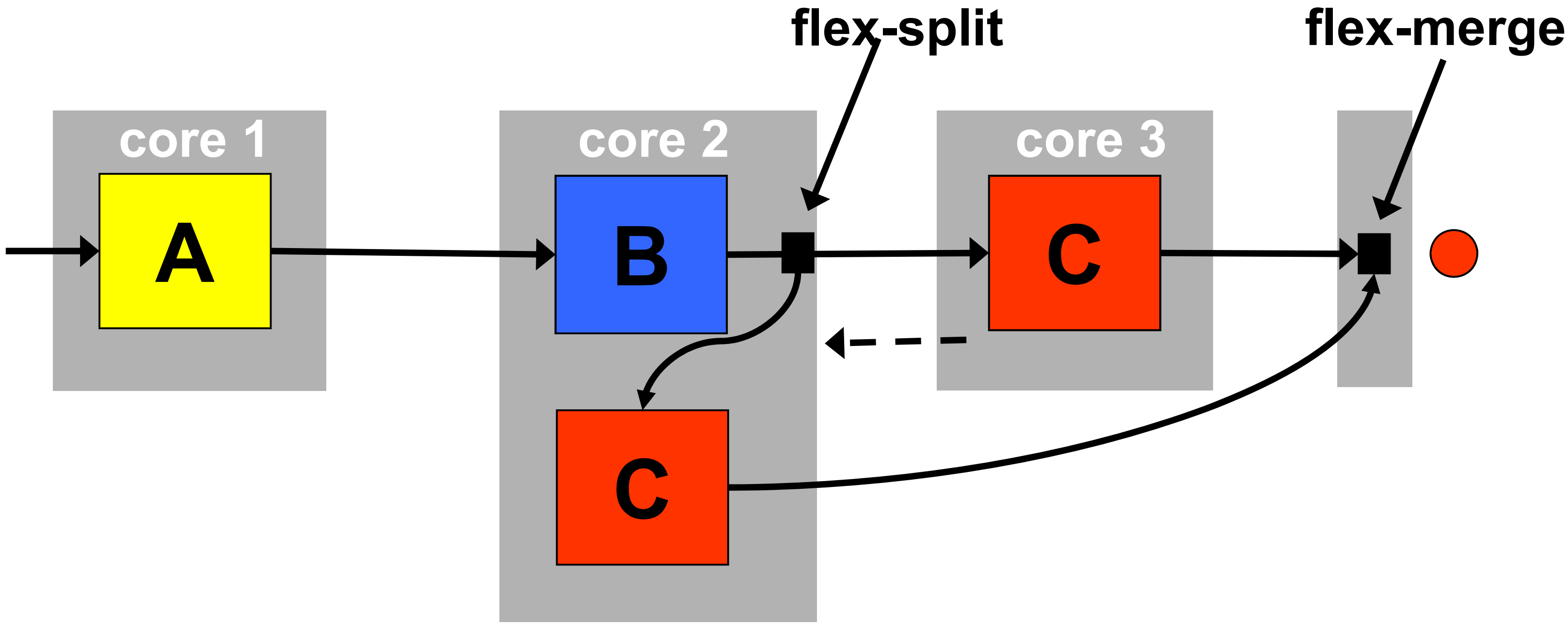

- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters




- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Our Solution: Flexible Filters




- Unused cycles on B are filled by working ahead on filter C with the data already present on B
- Push stream flow upstream of a bottleneck
- Semantic Preservation

# Related Works

- **Static Compiler Optimizations**
  - **StreamIt [Gordon et al, 2006]**
- **Dynamic Runtime Load Balancing**
  - **Work Stealing/Filter Migration**
    - **[Kakulavarapu et al., 2001]**
    - **Cilk [Frigo et al., 1998]**
    - **Flux [Shah et al., 2003]**
    - **Borealis [Xing et al., 2005]**
  - **Queue based load balancing**
    - **Diamond [Huston et al., 2005]** ***distributed search, queue based load balancing, filter re-ordering***
- **Combination Static+Dynamic**
  - **FlexStream [Hormati et al., 2009]** ***multiple competing programs***

# Outline

- Introduction
- Design Flow of a Stream Program with Flexibility
- Performance
- Implementation of Flexible Filters
- Experiments
  - CFAR Case Study

# Design Flow of a Stream Program with Flexible Filters

**Design stream algorithm**

↓

**Mapping**
- **Filters**
- **Memory**

↓

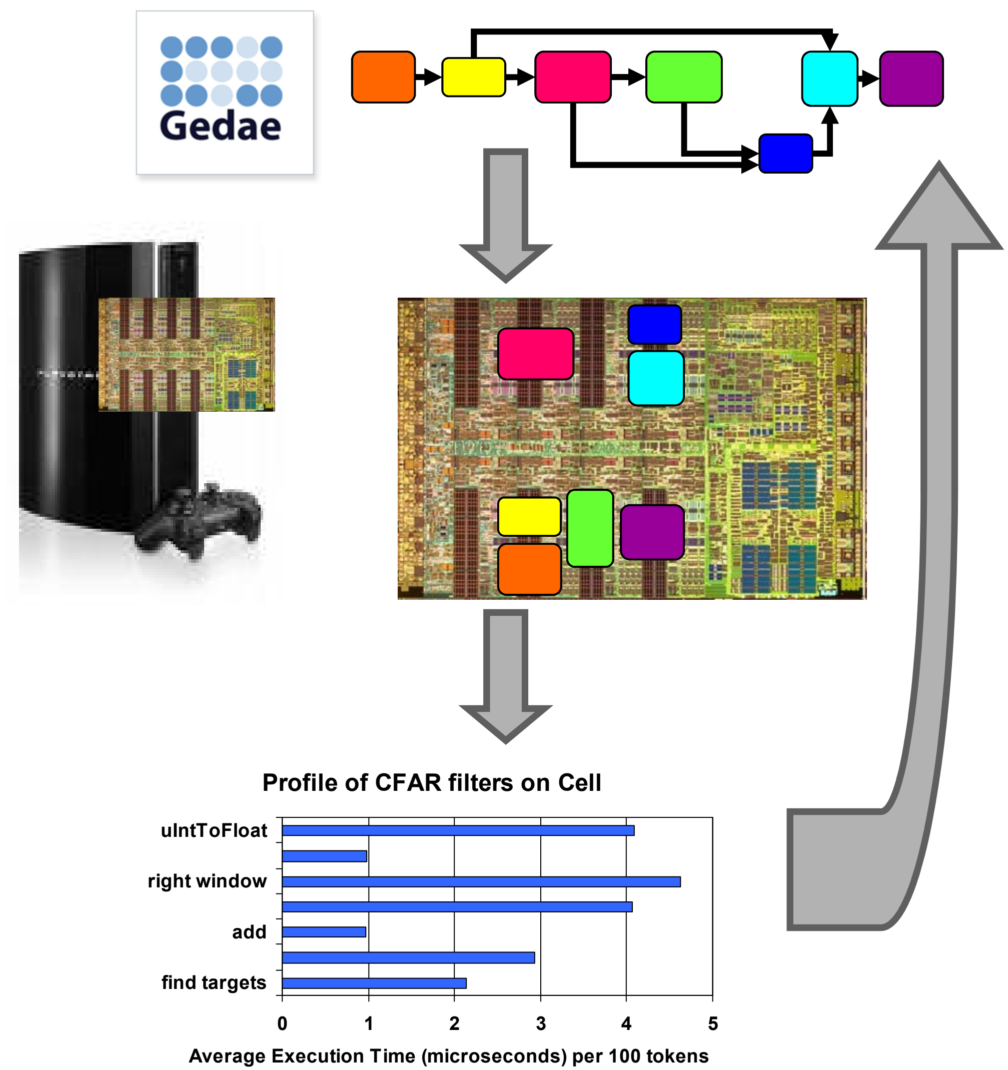
**Profile**

↓

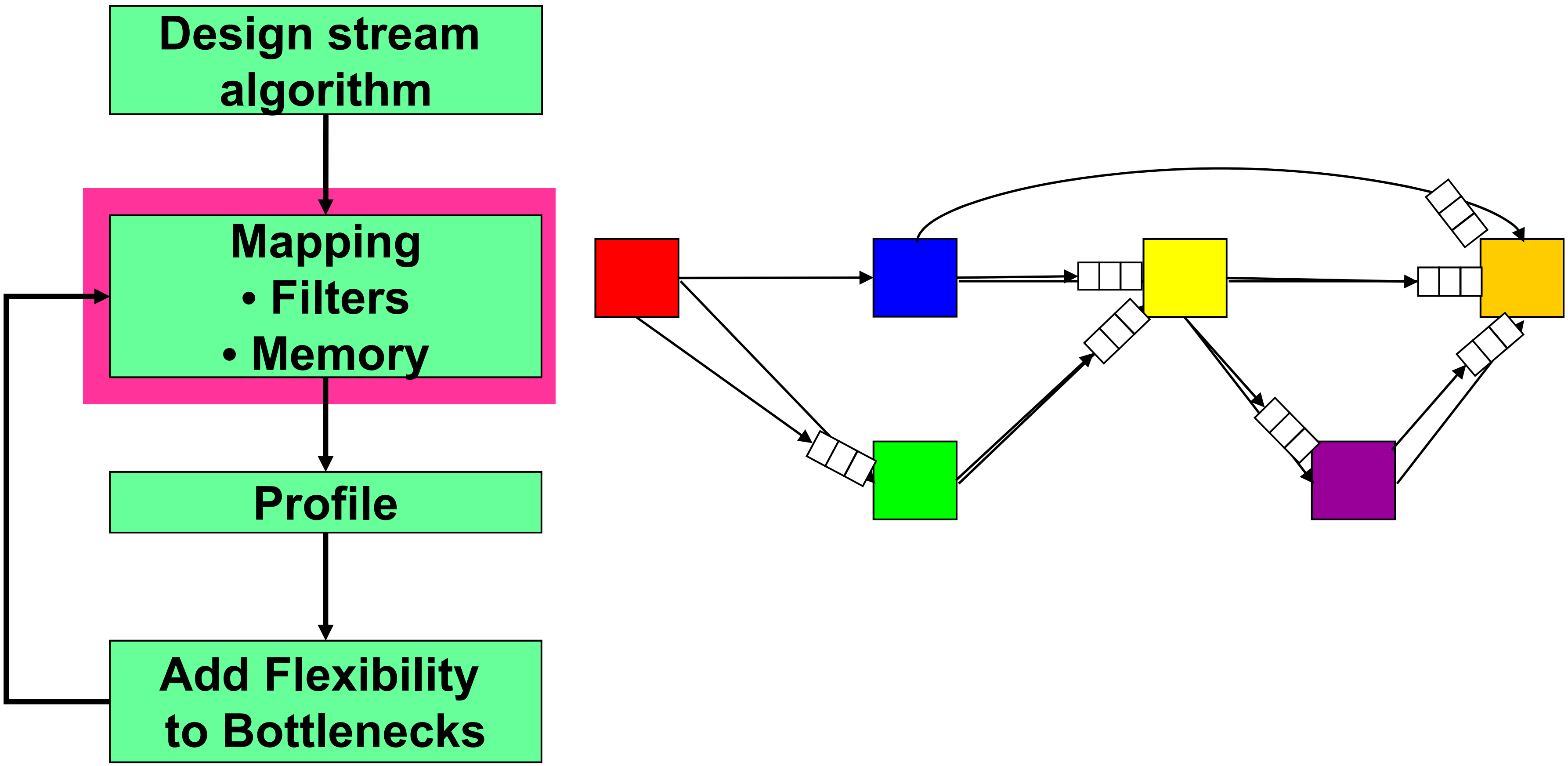
**Add Flexibility to Bottlenecks**

(loops back to Mapping)

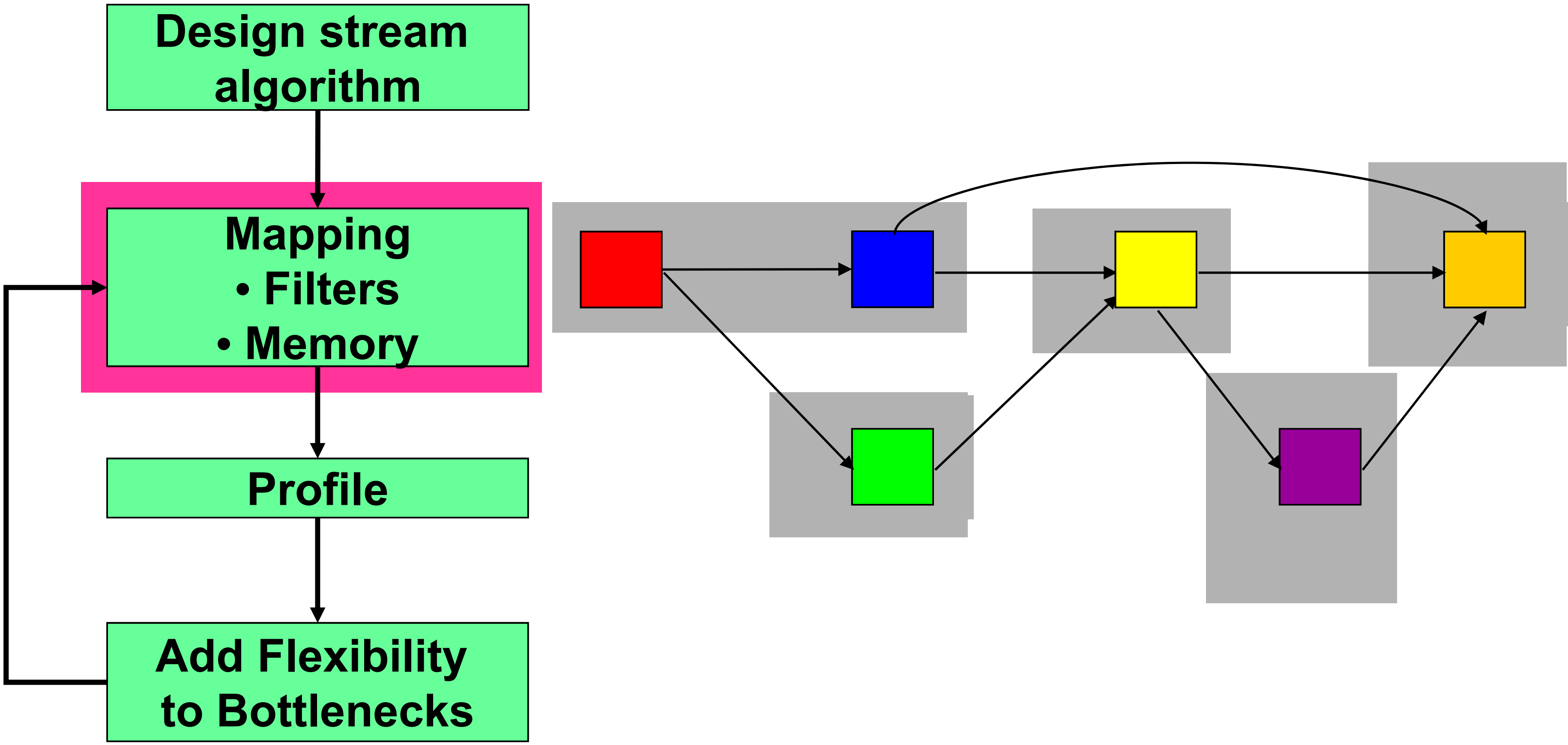
compiler/
design tools



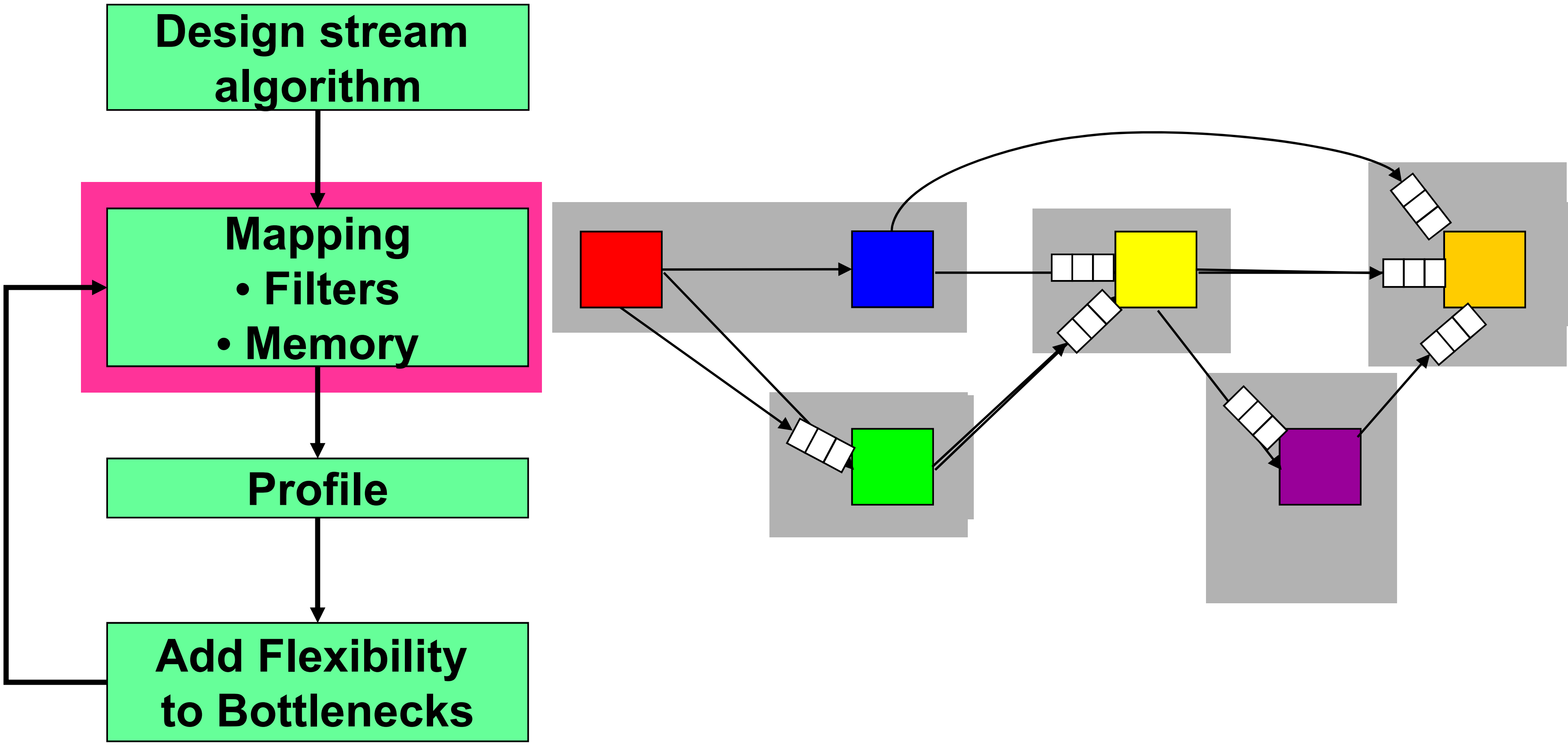
# Design Considerations



# Design Considerations



# Design Considerations



# Adding Flexibility

Design stream algorithm

↓

Mapping
- Filters
- Memory

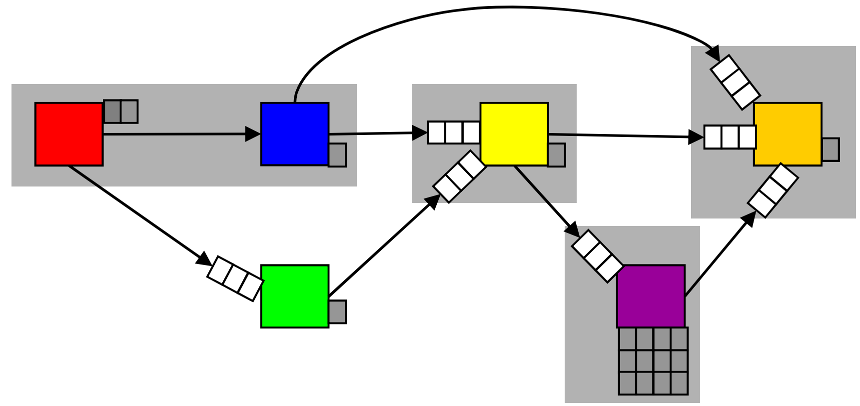
↓

Profile

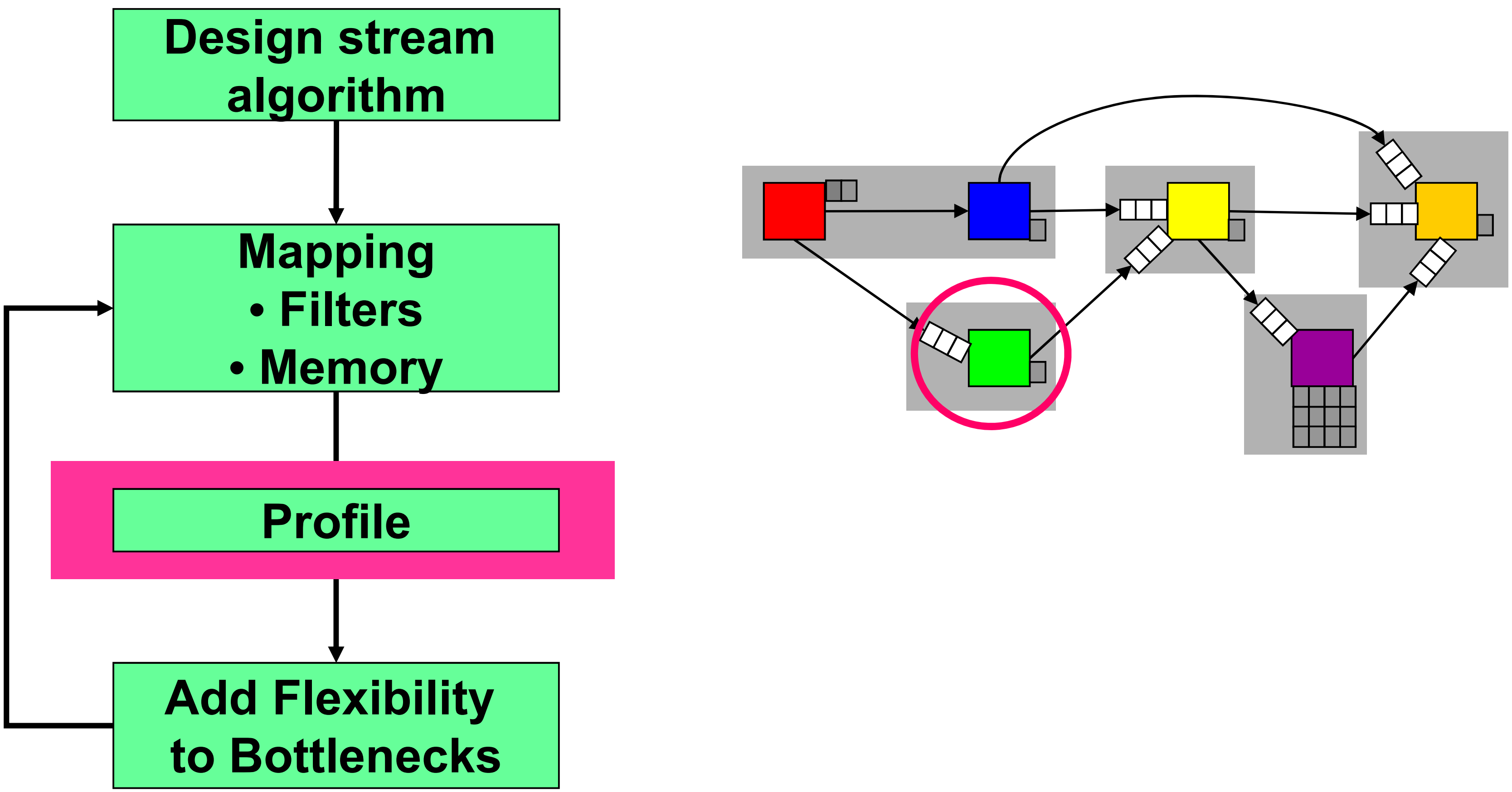
↓

Add Flexibility to Bottlenecks

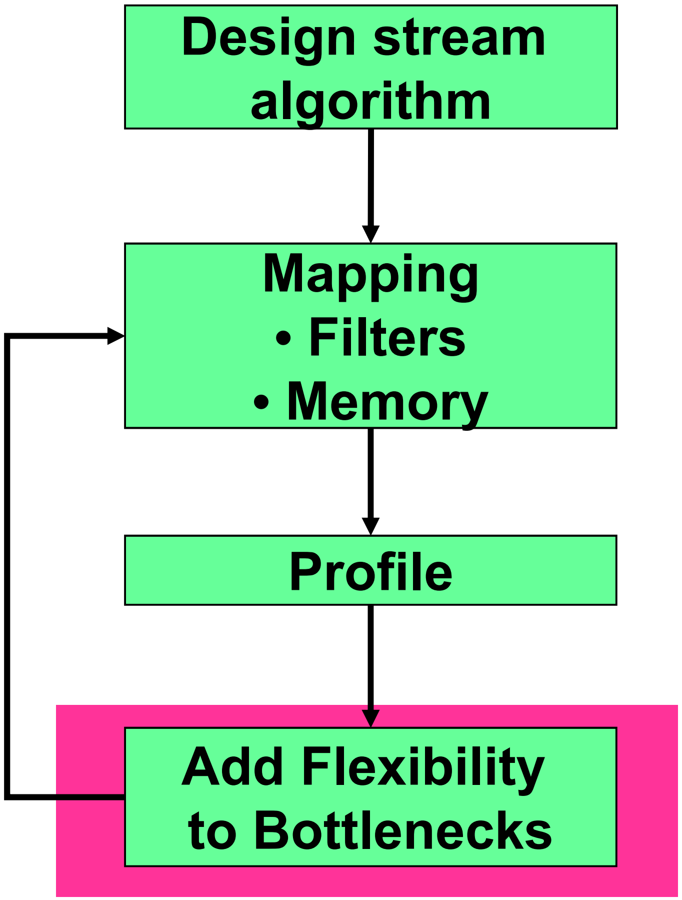
(→ back to Mapping)

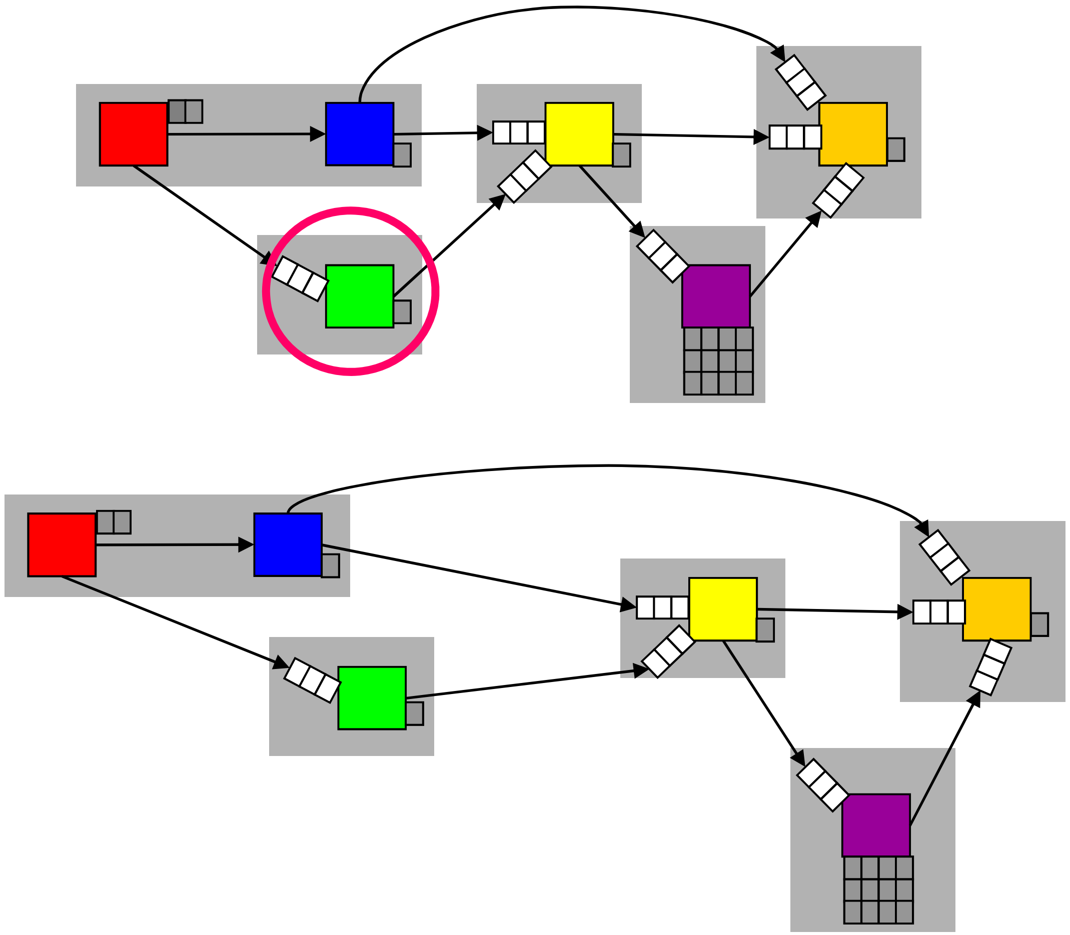


# Adding Flexibility

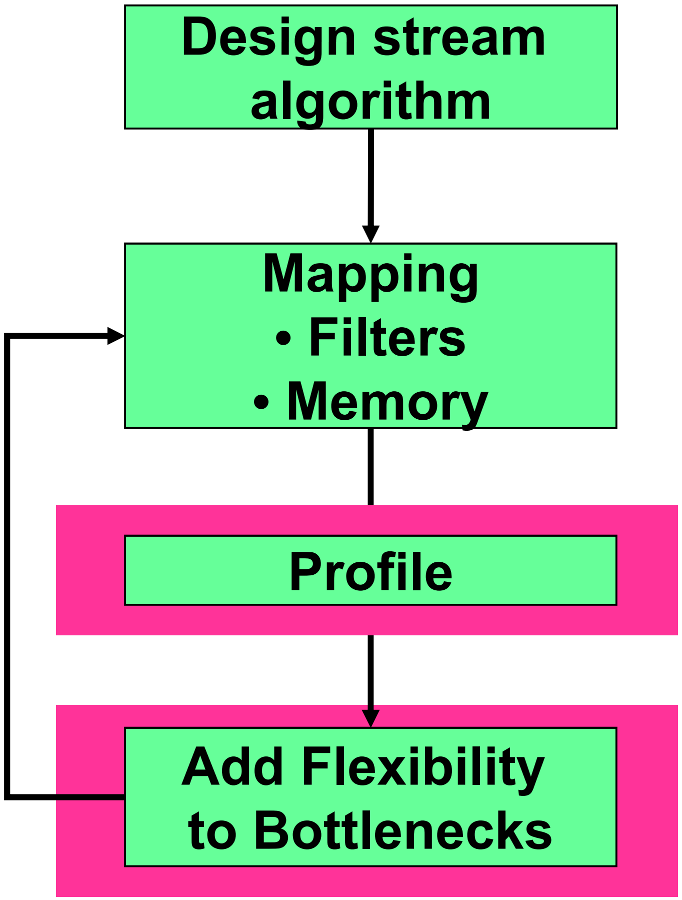


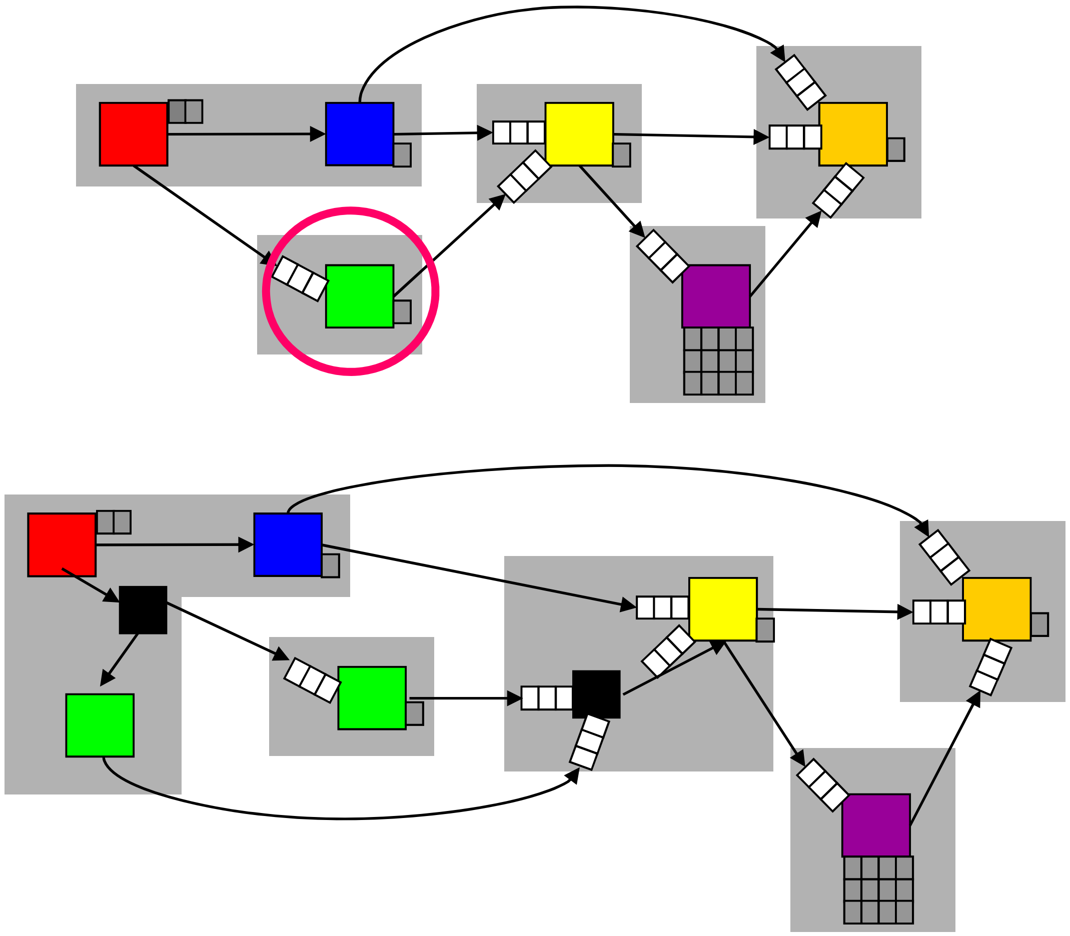
# Adding Flexibility



# Adding Flexibility

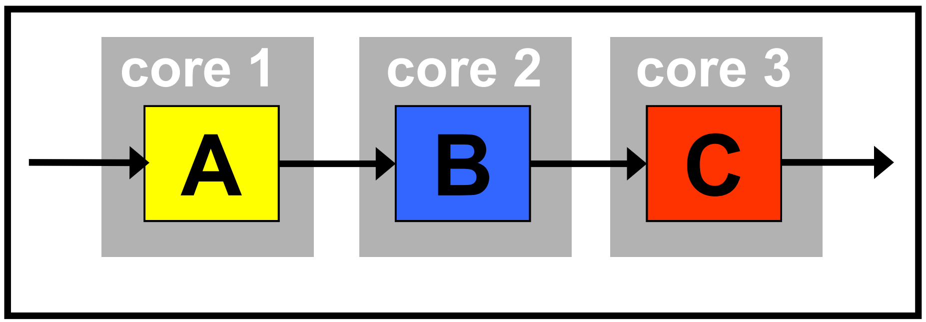


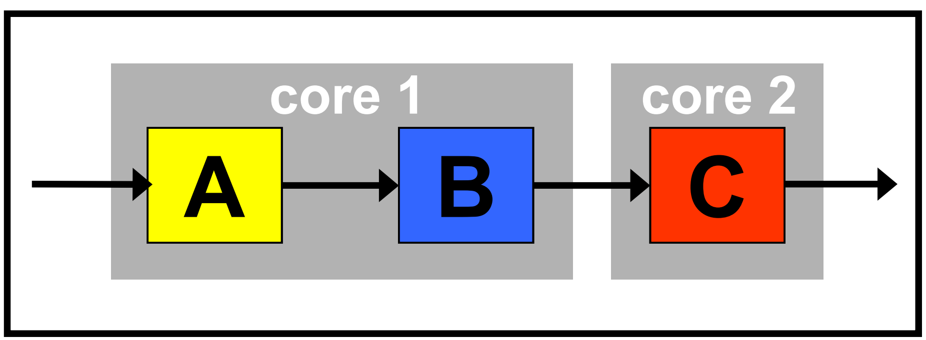
# Outline

- Introduction
- Design Flow of a Stream Program with Flexibility
- Performance
- Implementation of Flexible Filters
- Experiments
  - CFAR Case Study

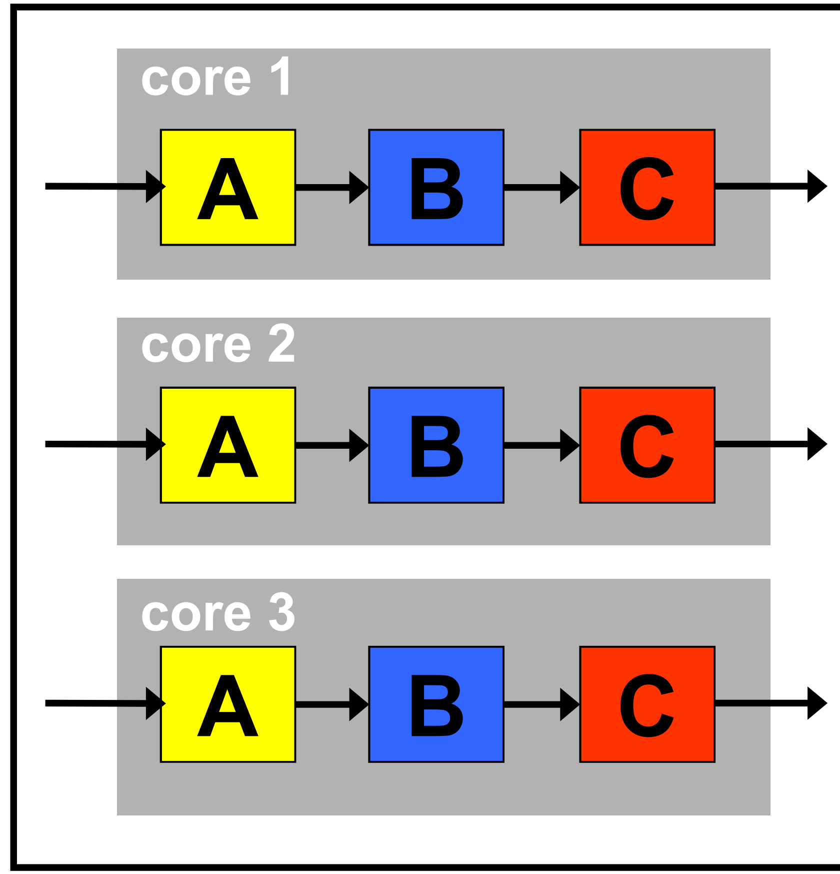
# Mapping Stream Programs to Multi-Core Platforms




*pipeline mapping*

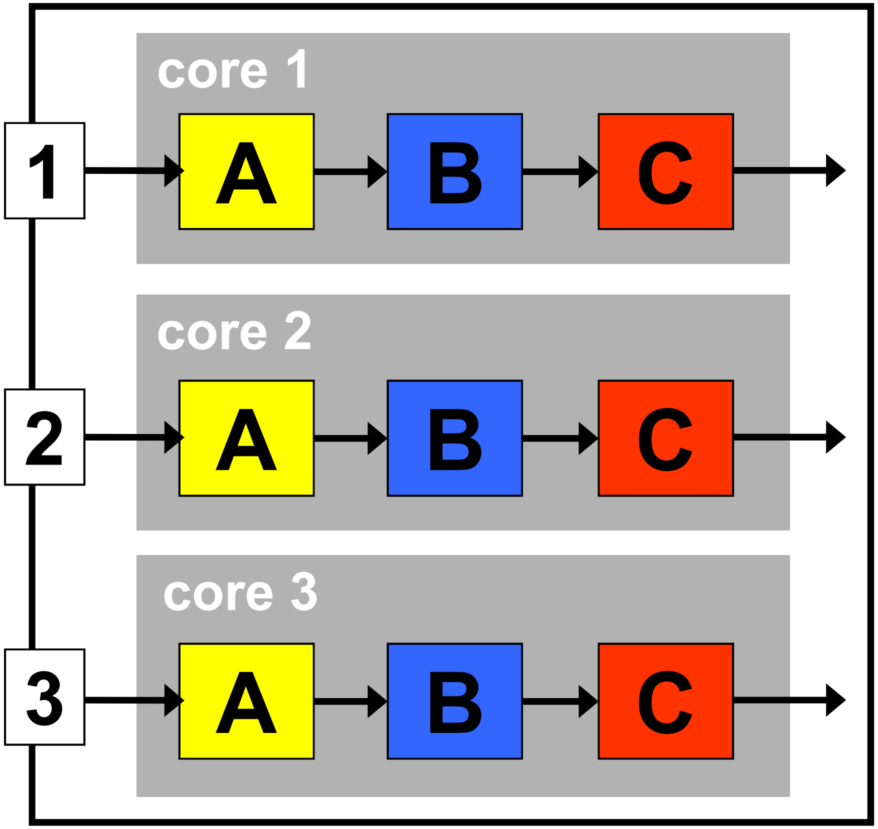



*sharing a core*




*SPMD mapping*

# Throughput: SPMD Mapping

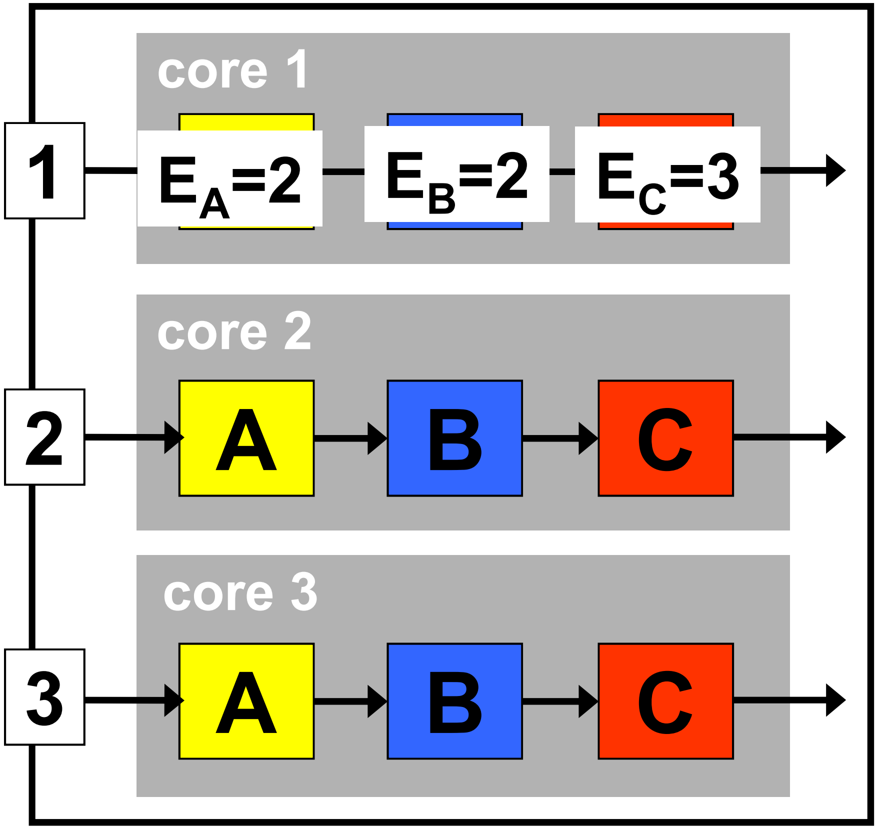



*SPMD mapping*

| | t0 | t1 | t2 | t3 | t4 | t5 | t6 |
|---|---|---|---|---|---|---|---|
| core 1 | | | | | | | |
| core 2 | | | | | | | |
| core 3 | | | | | | | |

3 tokens processed in 7 timesteps,
*ideal* throughput = 3/7 = 0.429

# Throughput: SPMD Mapping



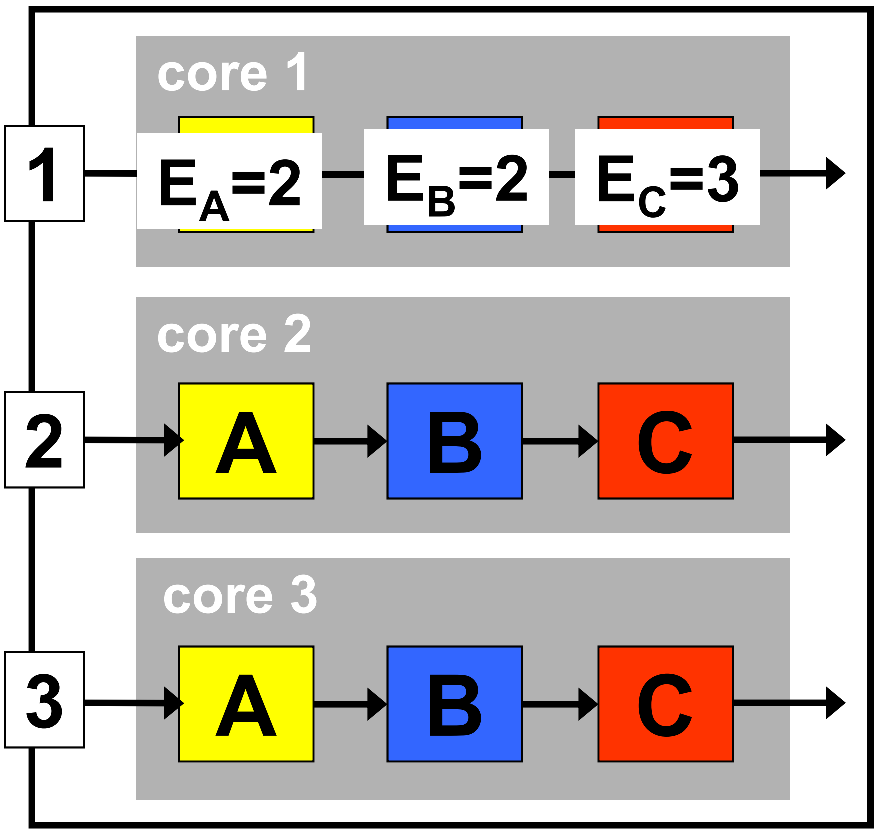

*SPMD mapping*

Suppose $E_A = 2$, $E_B = 2$, $E_C = 3$

| | t0 | t1 | t2 | t3 | t4 | t5 | t6 |
|---|---|---|---|---|---|---|---|
| core 1 | | | | | | | |
| core 2 | | | | | | | |
| core 3 | | | | | | | |

3 tokens processed in 7 timesteps,
*ideal* throughput = 3/7 = 0.429

# Throughput: SPMD Mapping



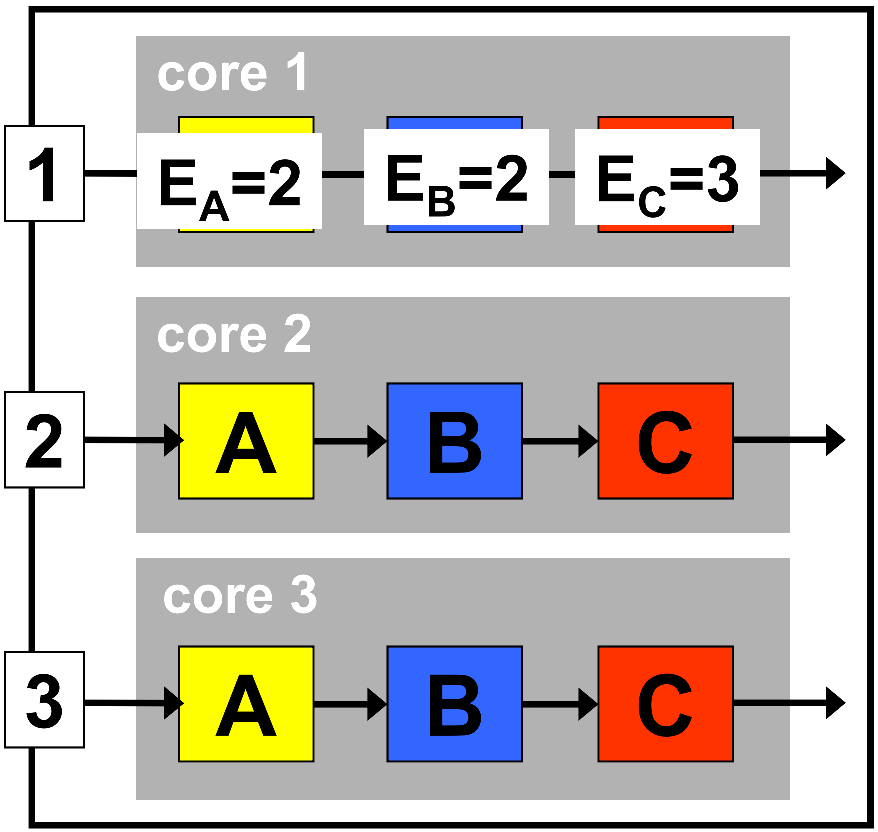

*SPMD mapping*

Suppose $E_A = 2$, $E_B = 2$, $E_C = 3$

| | t0 | t1 | t2 | t3 | t4 | t5 | t6 |
|---|---|---|---|---|---|---|---|
| core 1 | A1 | | | | | | |
| core 2 | A2 | | | | | | |
| core 3 | A3 | | | | | | |

3 tokens processed in 7 timesteps,
*ideal* throughput = 3/7 = 0.429

# Throughput: SPMD Mapping

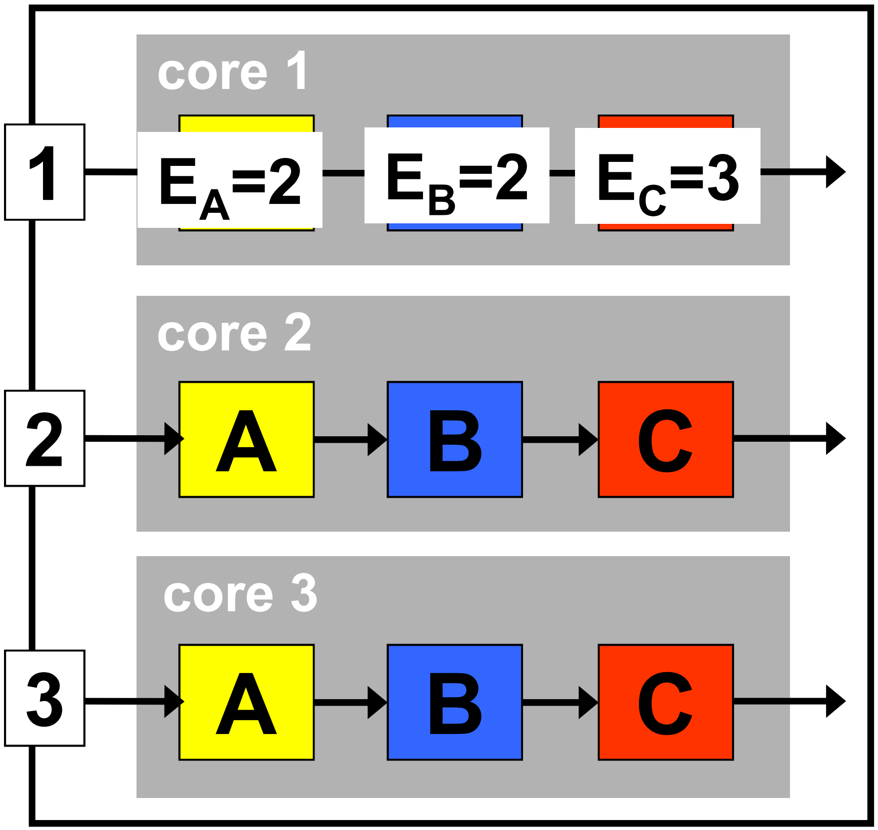



*SPMD mapping*

Suppose $E_A = 2$, $E_B = 2$, $E_C = 3$

| | t0 | t1 | t2 | t3 | t4 | t5 | t6 |
|---|---|---|---|---|---|---|---|
| core 1 | A1 | | B1 | | | | |
| core 2 | A2 | | B2 | | | | |
| core 3 | A3 | | B3 | | | | |

# Throughput: SPMD Mapping



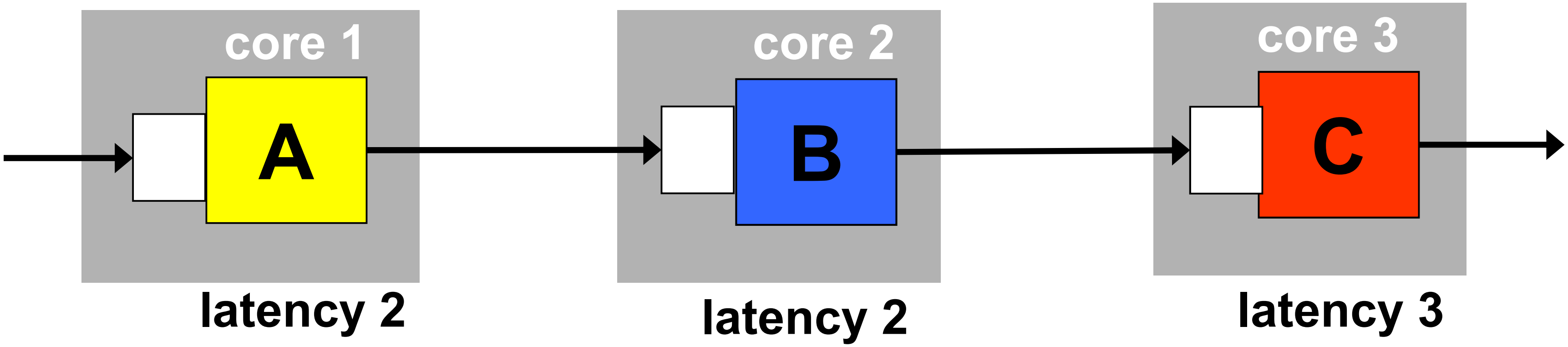

*SPMD mapping*

**Suppose $E_A = 2$, $E_B = 2$, $E_C = 3$**

| | t0 | t1 | t2 | t3 | t4 | t5 | t6 |
|---|---|---|---|---|---|---|---|
| core 1 | A1 | | B1 | | C1 | | |
| core 2 | A2 | | B2 | | C2 | | |
| core 3 | A3 | | B3 | | C3 | | |

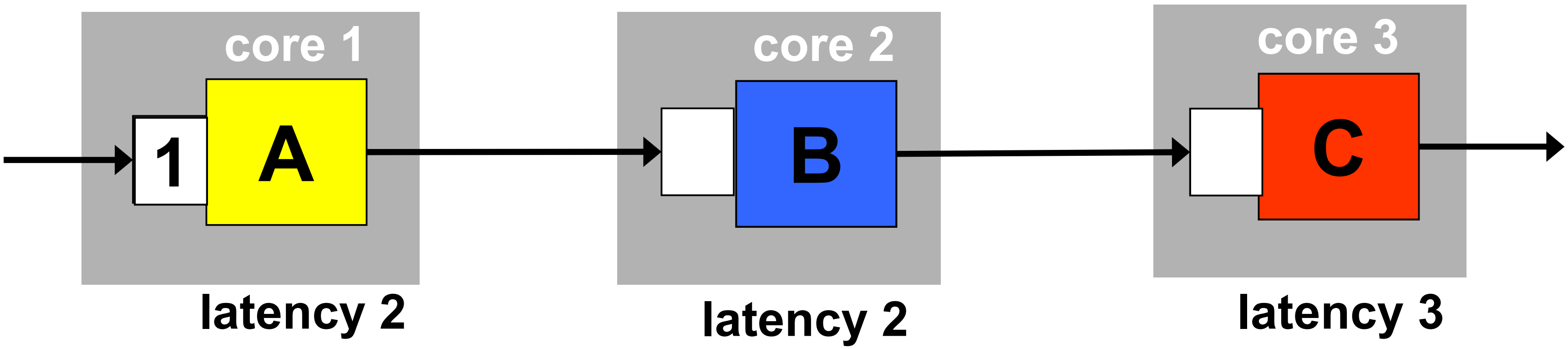
**3 tokens processed in 7 timesteps,**
***ideal* throughput = 3/7 = 0.429**

# Throughput: Pipeline Mapping



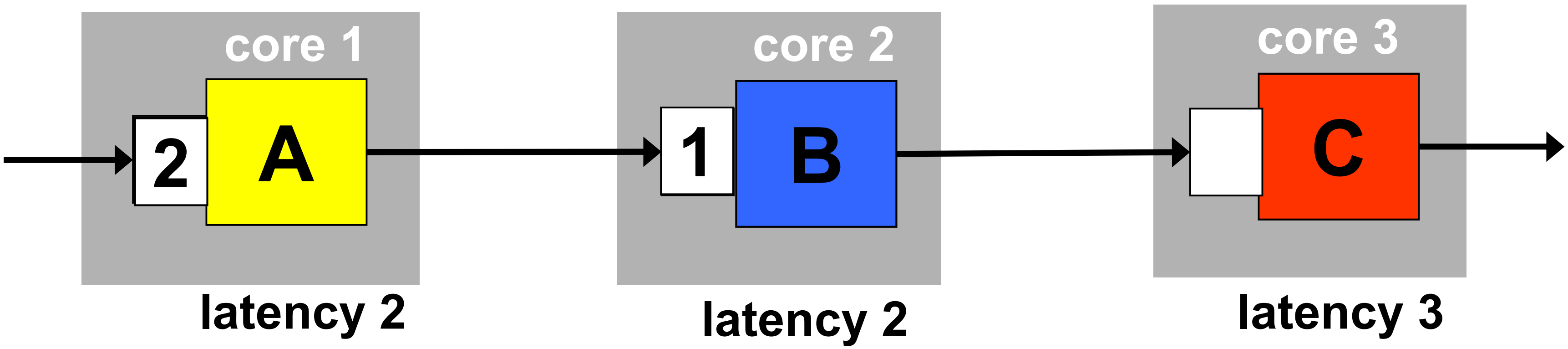

| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 |
|---|---|---|---|---|---|---|---|---|---|---|
| core 1 | | | | | | | | | | |
| core 2 | | | | | | | | | | |
| core 3 | | | | | | | | | | |

# Throughput: Pipeline Mapping



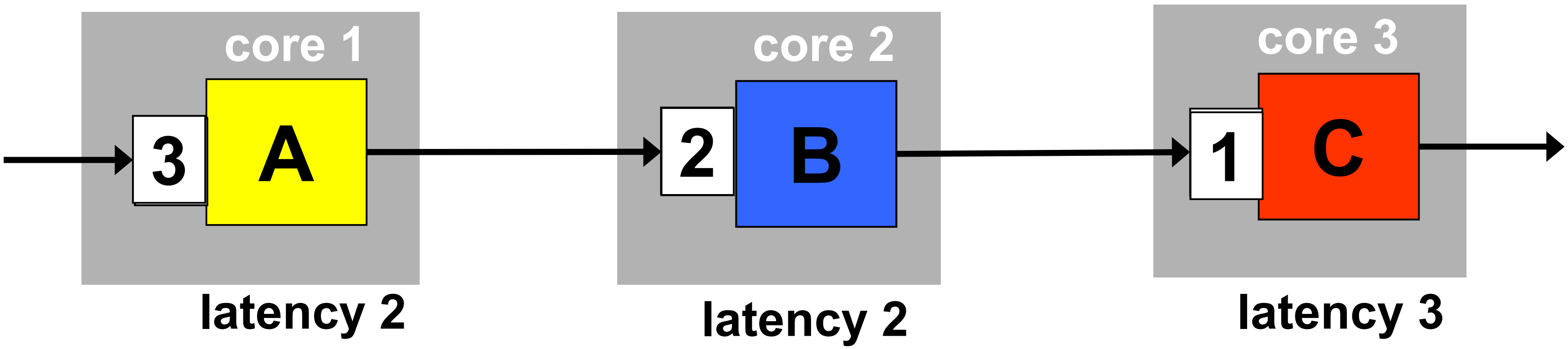

| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 |
|---|---|---|---|---|---|---|---|---|---|---|
| core 1 | A1 | | | | | | | | | |
| core 2 | | | | | | | | | | |
| core 3 | | | | | | | | | | |

# Throughput: Pipeline Mapping



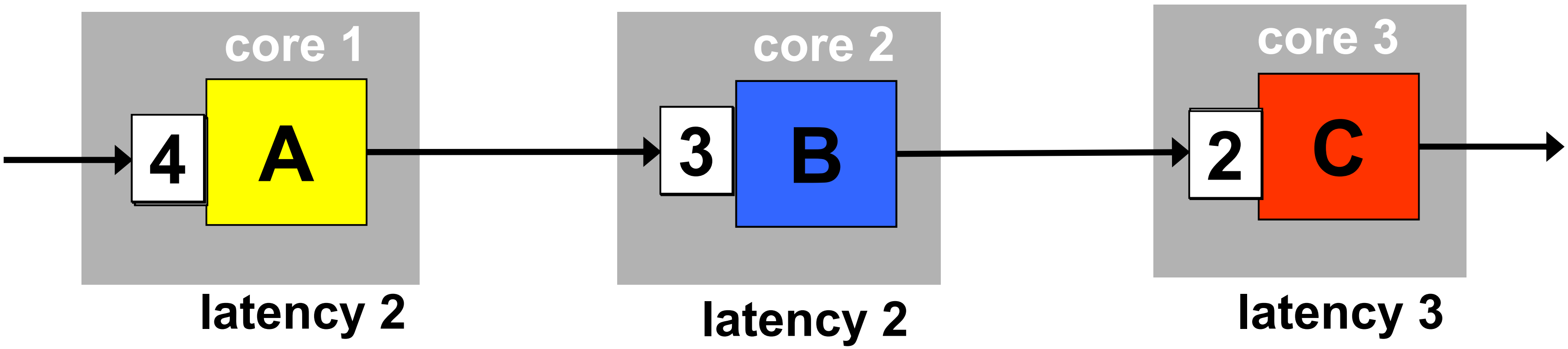

| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 |
|---|---|---|---|---|---|---|---|---|---|---|
| core 1 | A1 | | A2 | | | | | | | |
| core 2 | | | B1 | | | | | | | |
| core 3 | | | | | | | | | | |

# Throughput: Pipeline Mapping




| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 |
|---|---|---|---|---|---|---|---|---|---|---|
| core 1 | A1 | | A2 | | A3 | | | | | |
| core 2 | | | B1 | | B2 | | | | | |
| core 3 | | | | | C1 | | | | | |

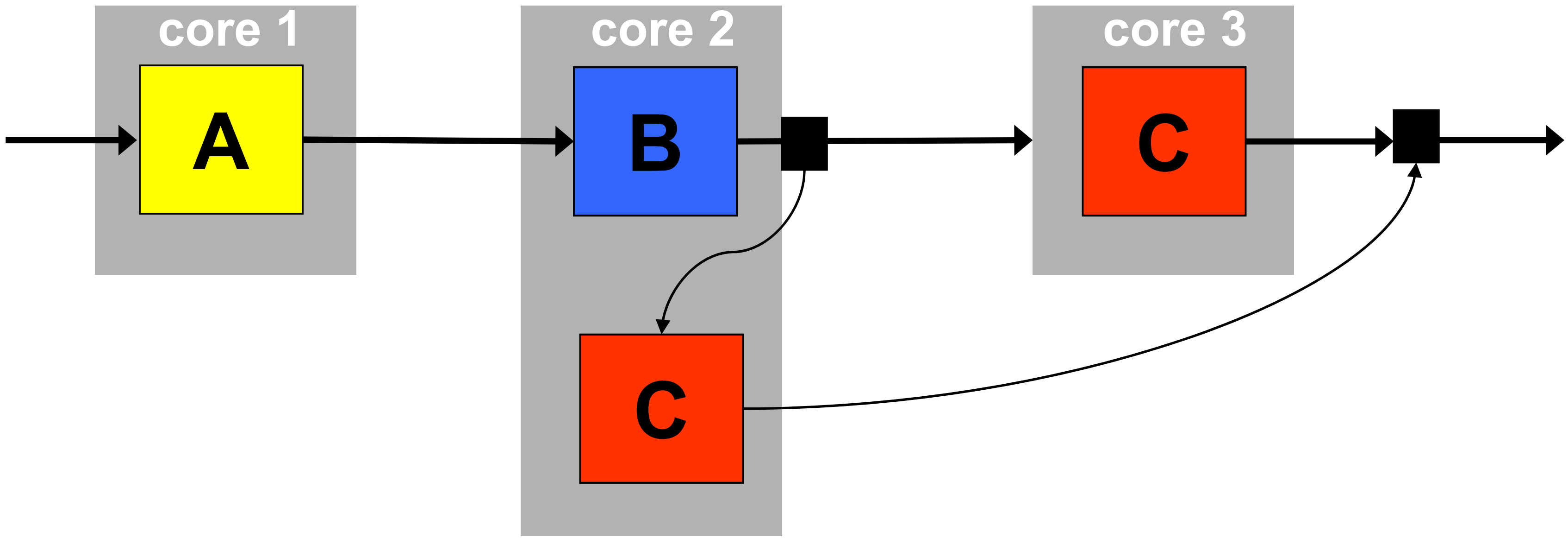
# Throughput: Pipeline Mapping




| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 |
|---|---|---|---|---|---|---|---|---|---|---|
| core 1 | A1 | | A2 | | A3 | | | A4 | | |
| core 2 | | | B1 | | B2 | | | B3 | | |
| core 3 | | | | | C1 | | | C2 | | |

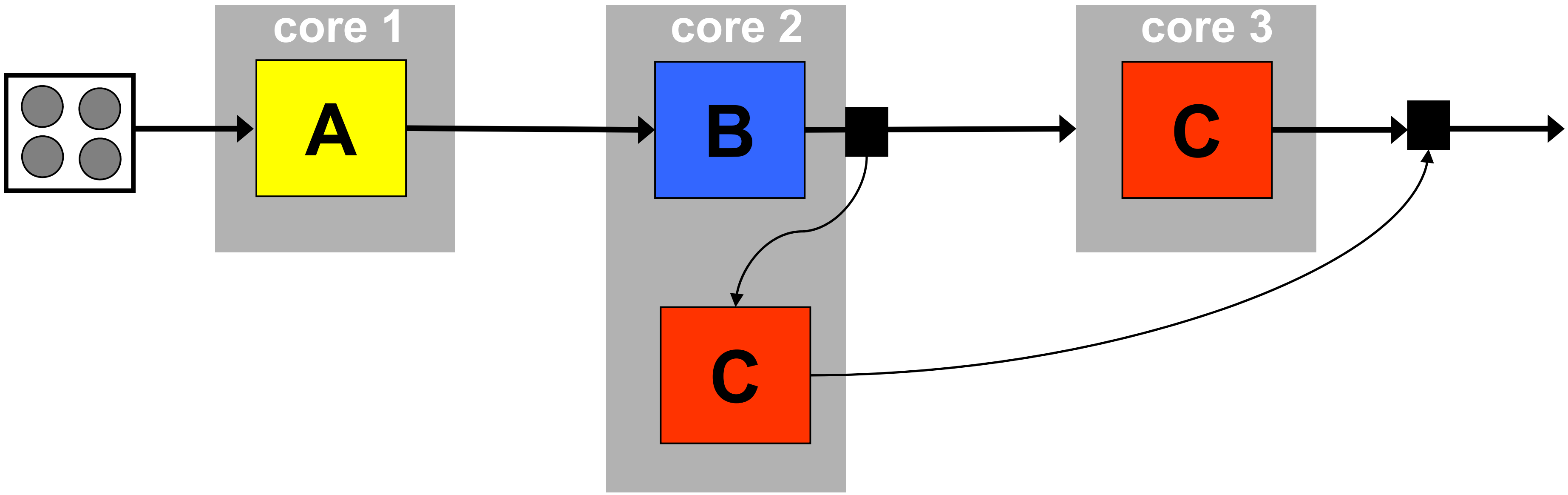
throughput = 1/3 = 0.333 < 0.429

# Data Blocks



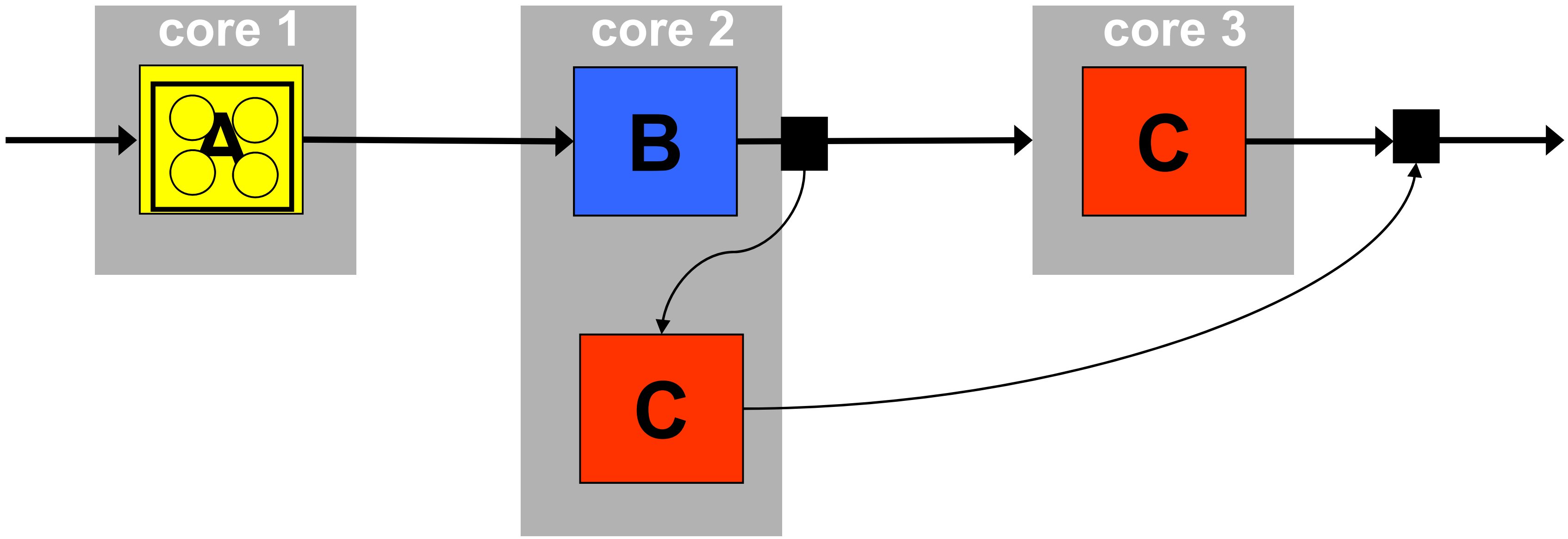

**data block: group of data tokens**

# Data Blocks



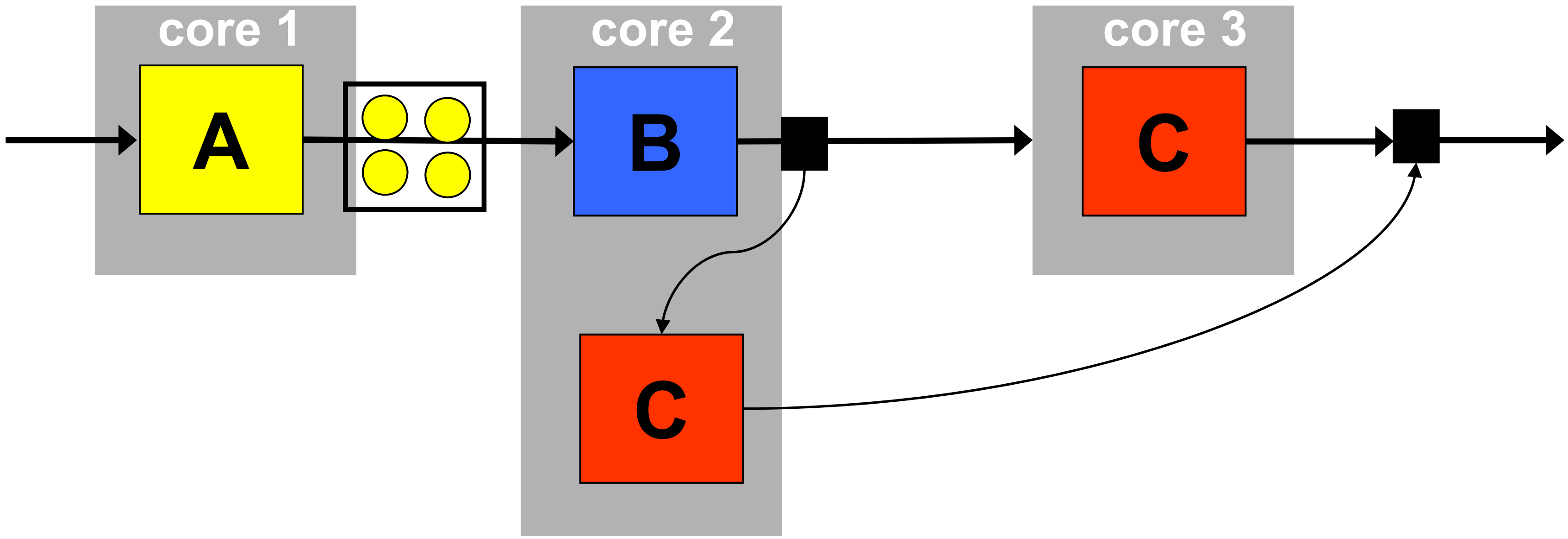

data block: group of data tokens

# Data Blocks



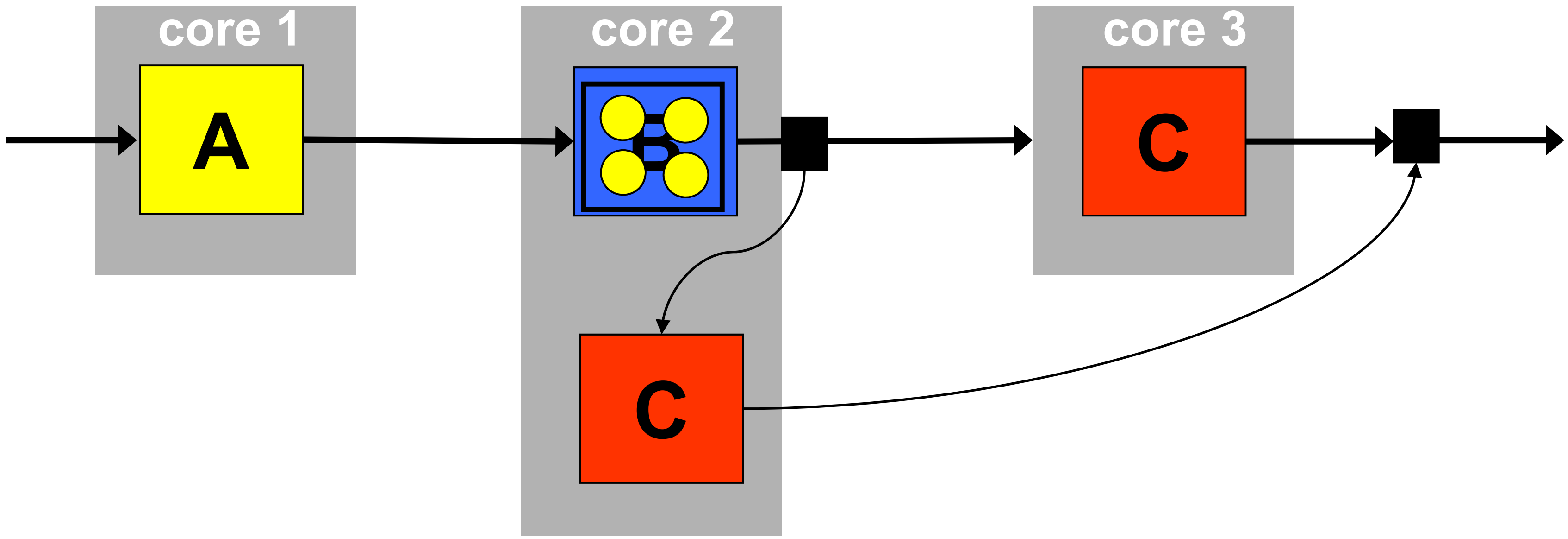

**data block**: group of data tokens

# Data Blocks



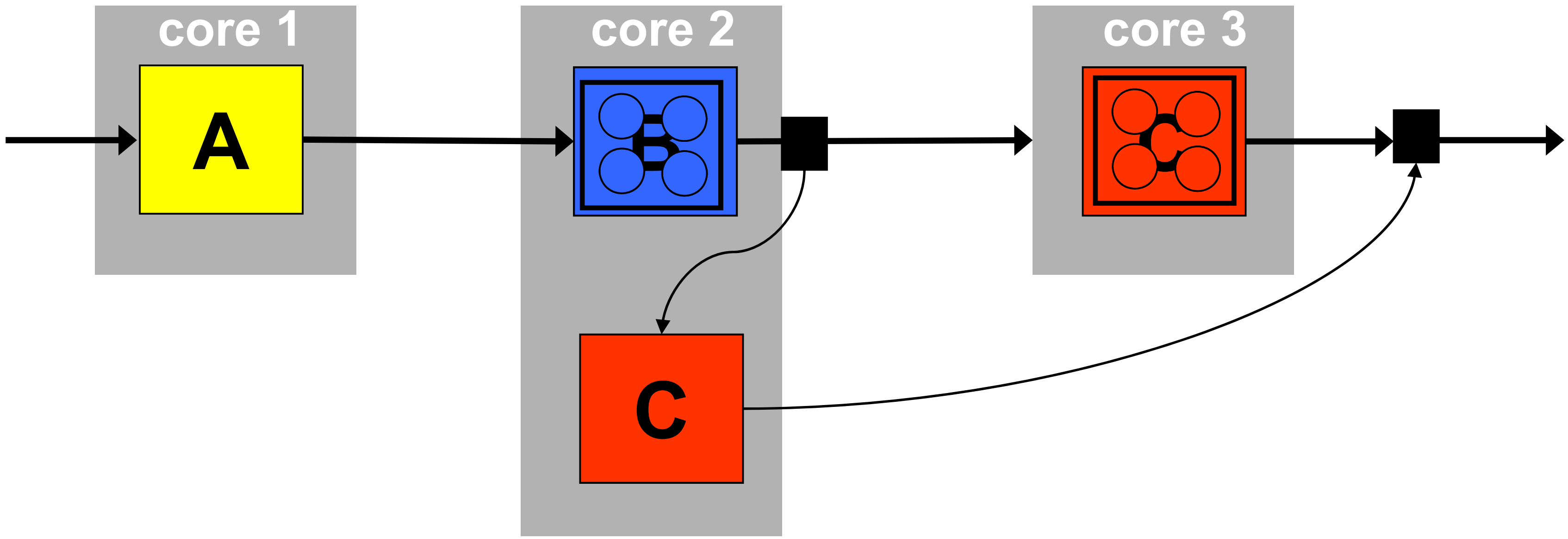

**data block: group of data tokens**

# Data Blocks



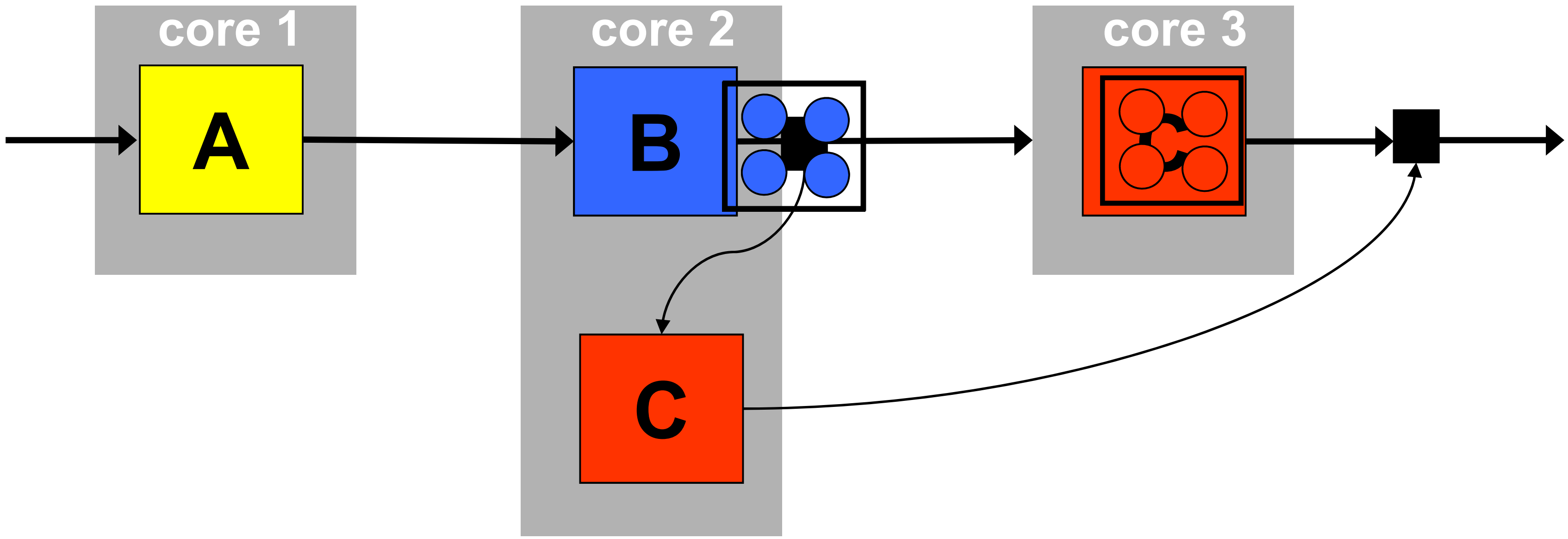

data block: group of data tokens

# Data Blocks



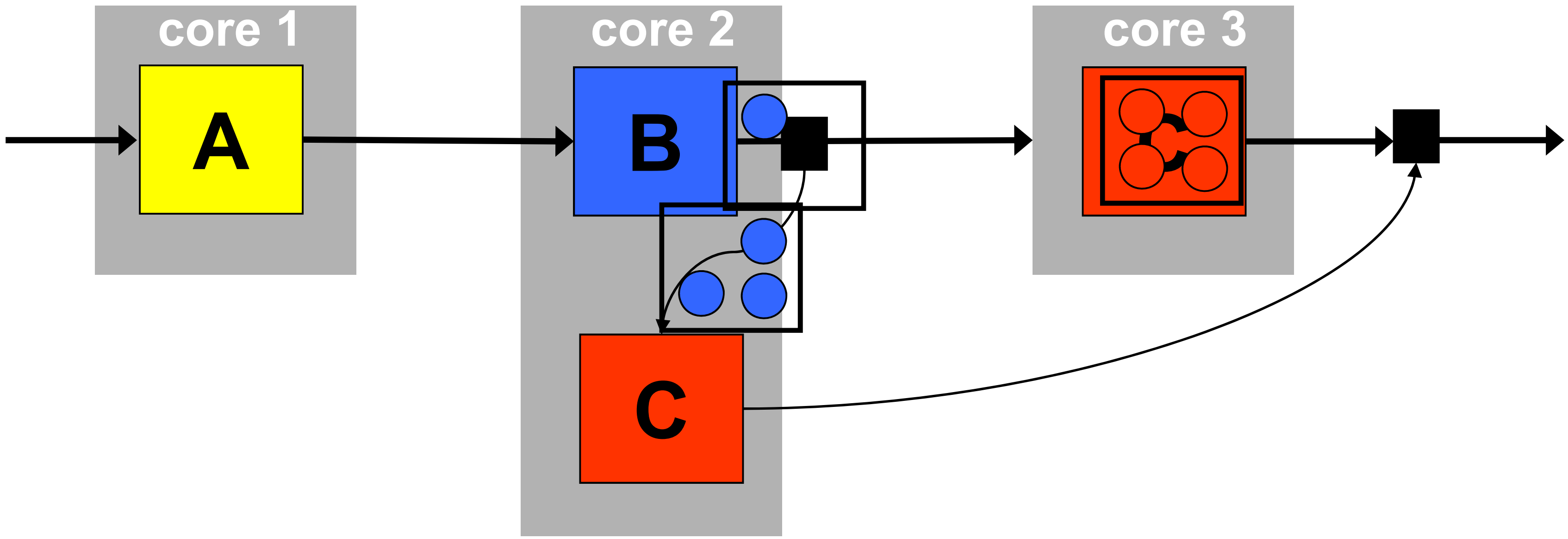

data block: group of data tokens

# Data Blocks



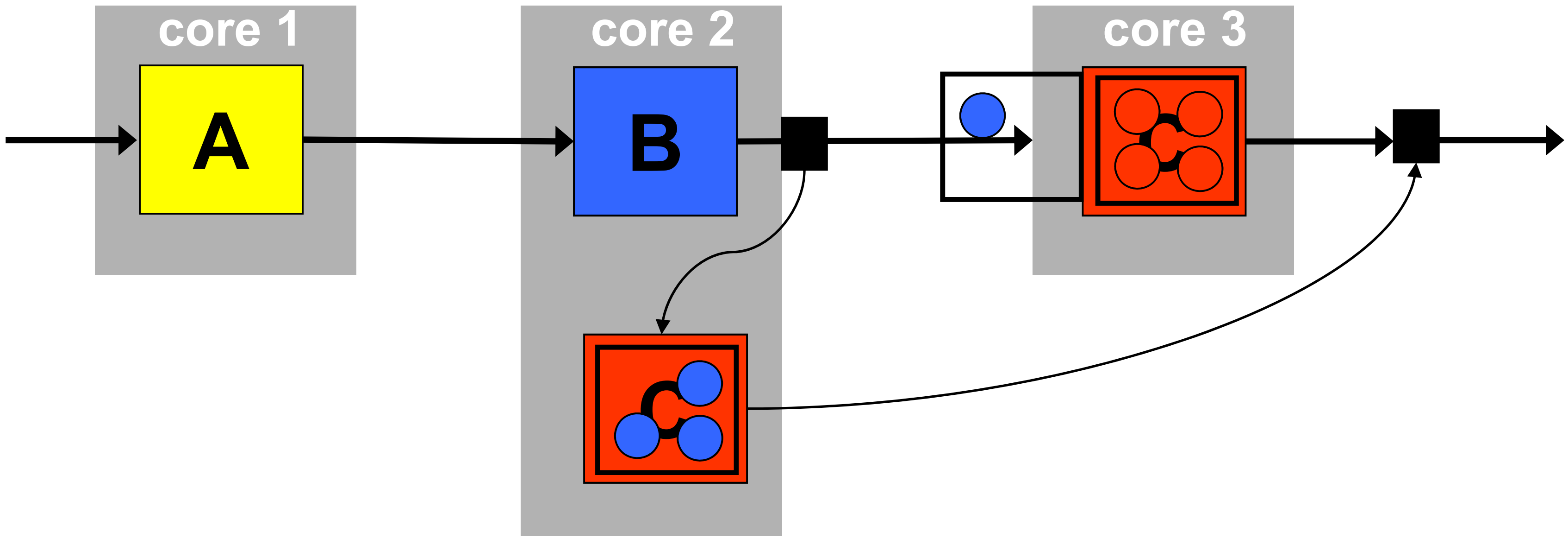

**data block**: group of data tokens

# Data Blocks



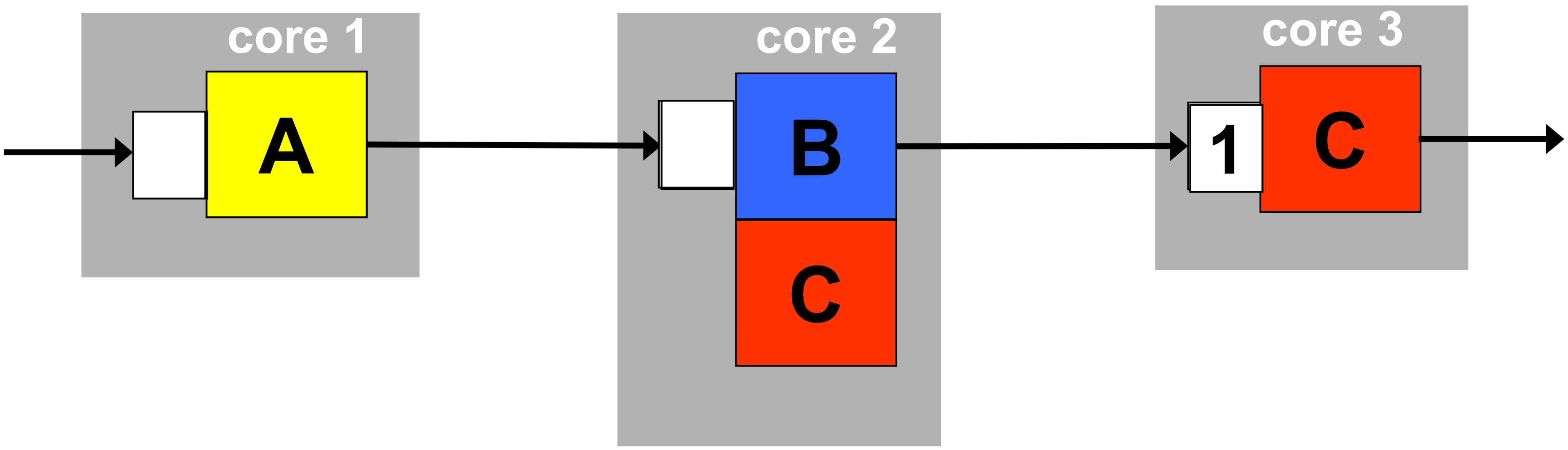

**data block**: group of data tokens

# Data Blocks




**data block**: group of data tokens

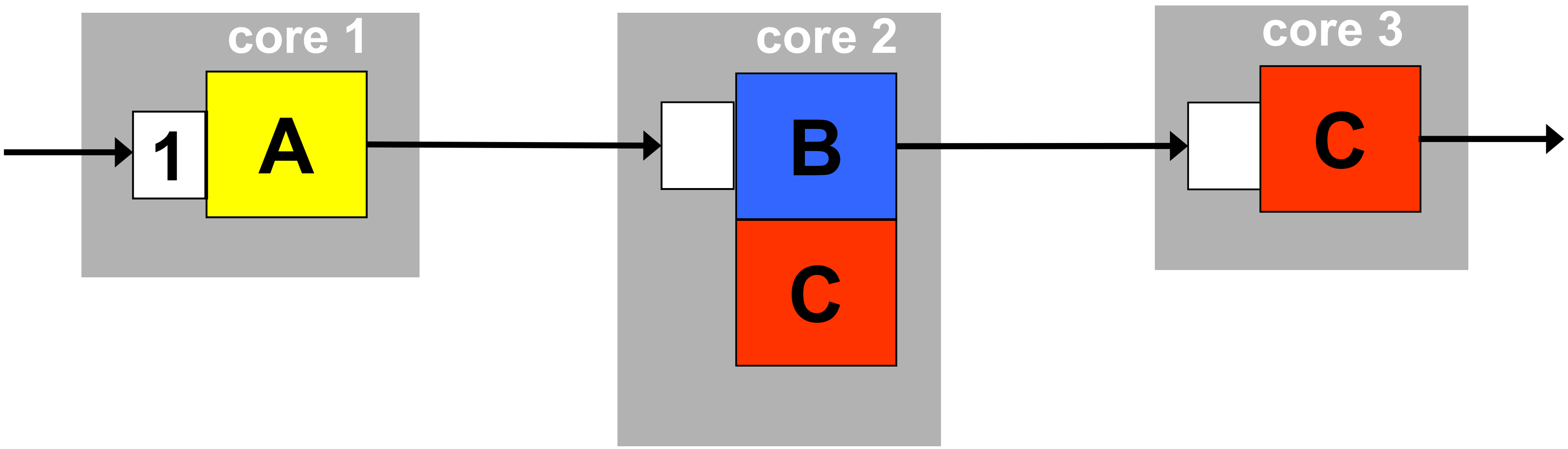
# Throughput: Pipeline Augmented with Flexibility




| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 |
|---|---|---|---|---|---|---|---|---|---|---|
| core 1 | | | | | | | | | | |
| core 2 | | | | | | | | | | |
| core 3 | | | | | | | | | | |

throughput = 2/5 = 0.4 < 0.429 (but > 0.333)

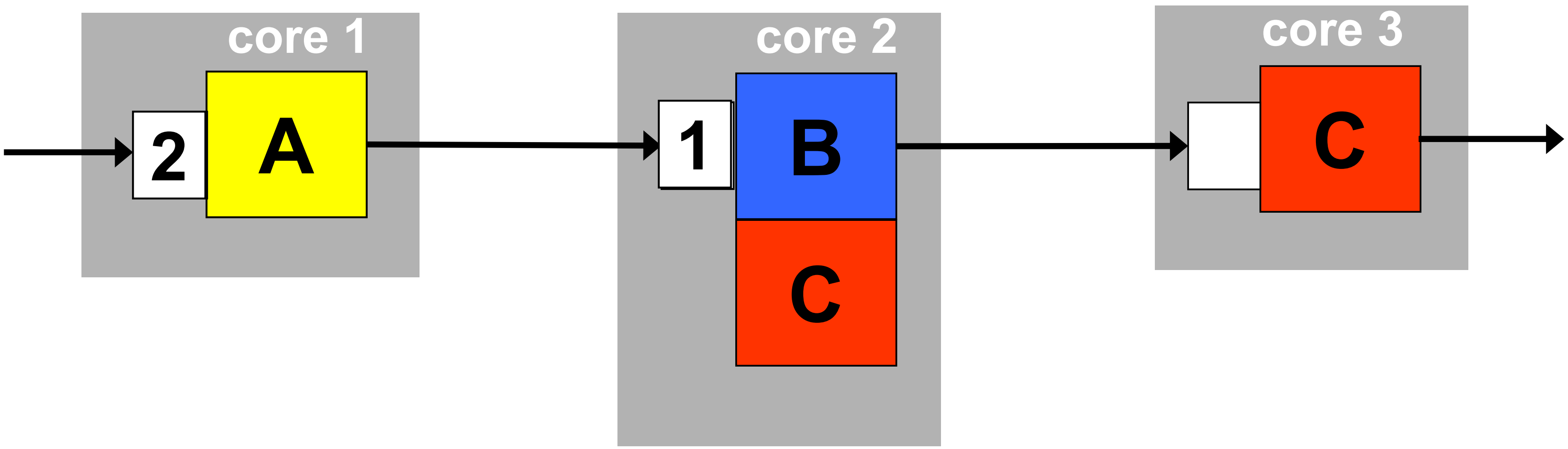
# Throughput: Pipeline Augmented with Flexibility




| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 |
|---|---|---|---|---|---|---|---|---|---|---|
| core 1 | A1 | | | | | | | | | |
| core 2 | | | | | | | | | | |
| core 3 | | | | | | | | | | |

**throughput = 2/5 = 0.4 < 0.429 (but > 0.333)**

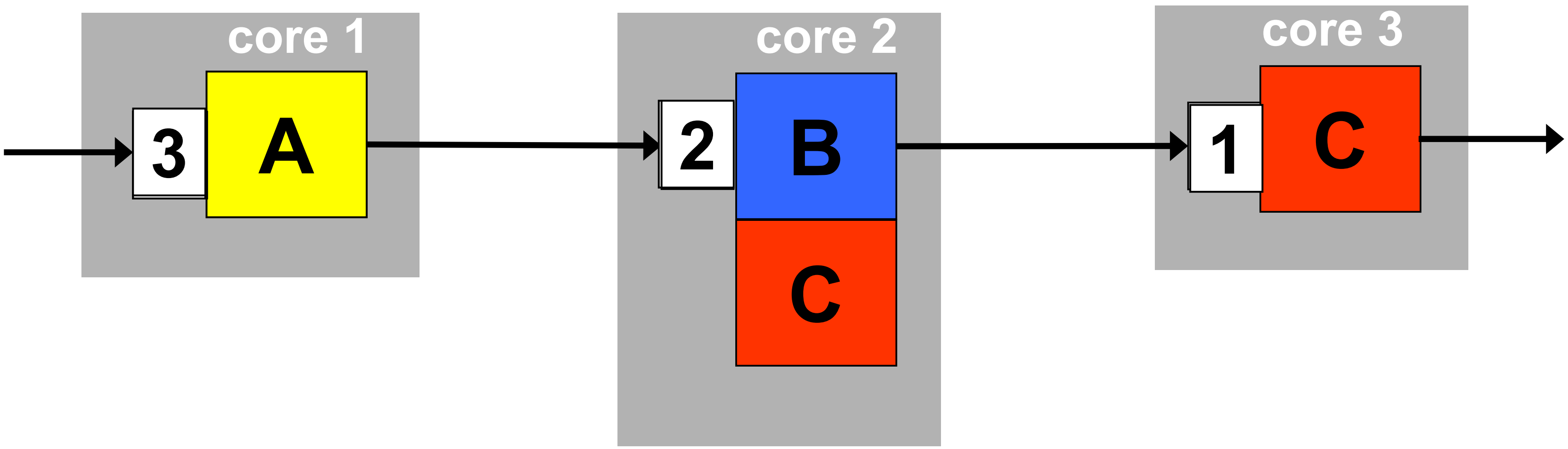
# Throughput: Pipeline Augmented with Flexibility




| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 |
|---|---|---|---|---|---|---|---|---|---|---|
| core 1 | A1 | | A2 | | | | | | | |
| core 2 | | | B1 | | | | | | | |
| core 3 | | | | | | | | | | |

throughput = 2/5 = 0.4 < 0.429 (but > 0.333)

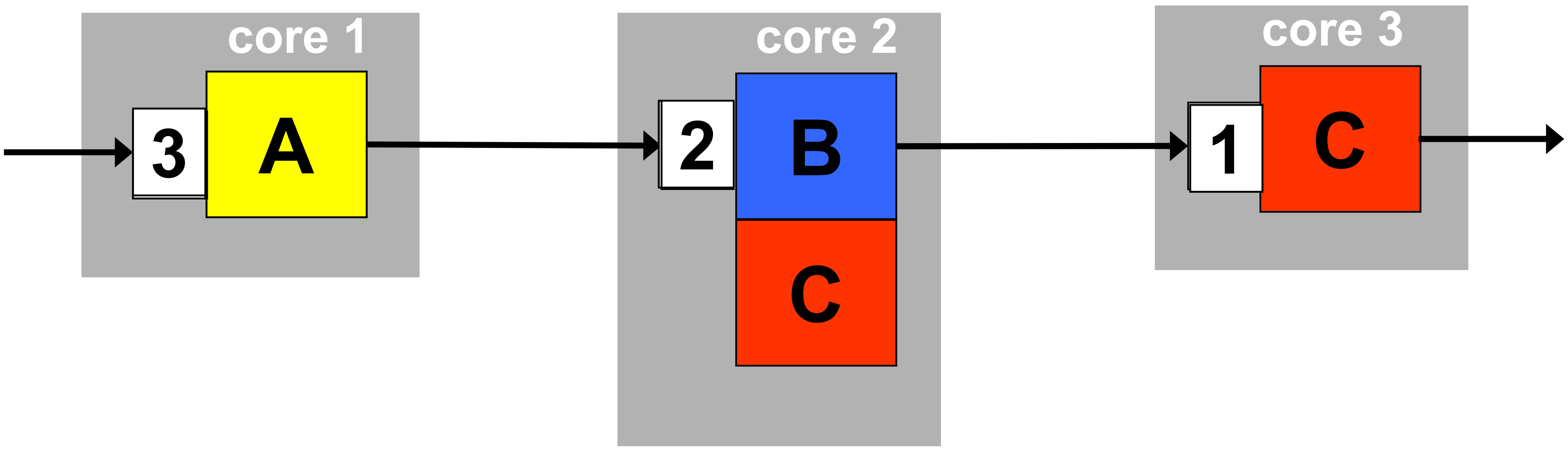
# Throughput: Pipeline Augmented with Flexibility




| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 |
|---|---|---|---|---|---|---|---|---|---|---|
| core 1 | A1 | | A2 | | A3 | | | | | |
| core 2 | | | B1 | | B2 | | | | | |
| core 3 | | | | | C1 | | | | | |

throughput = 2/5 = 0.4 < 0.429 (but > 0.333)

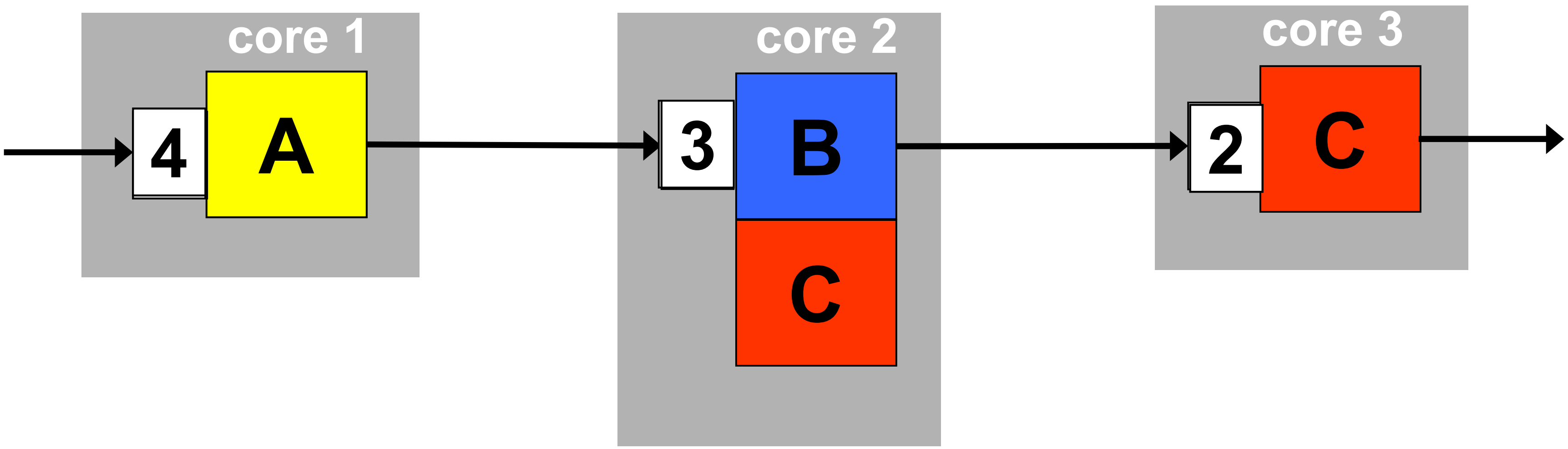
# Throughput: Pipeline Augmented with Flexibility




| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 |
|---|---|---|---|---|---|---|---|---|---|---|
| core 1 | A1 | | A2 | | A3 | | | | | |
| core 2 | | | B1 | | B2 | | C2 | | | |
| core 3 | | | | | C1 | | | | | |

throughput = 2/5 = 0.4 < 0.429 (but > 0.333)

# Throughput: Pipeline Augmented with Flexibility




| | t0 | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 |
|---|---|---|---|---|---|---|---|---|---|---|
| core 1 | A1 | | A2 | | A3 | | | A4 | | |
| core 2 | | | B1 | | B2 | | C2 | B3 | | |
| core 3 | | | | | C1 | | | C2 | | |

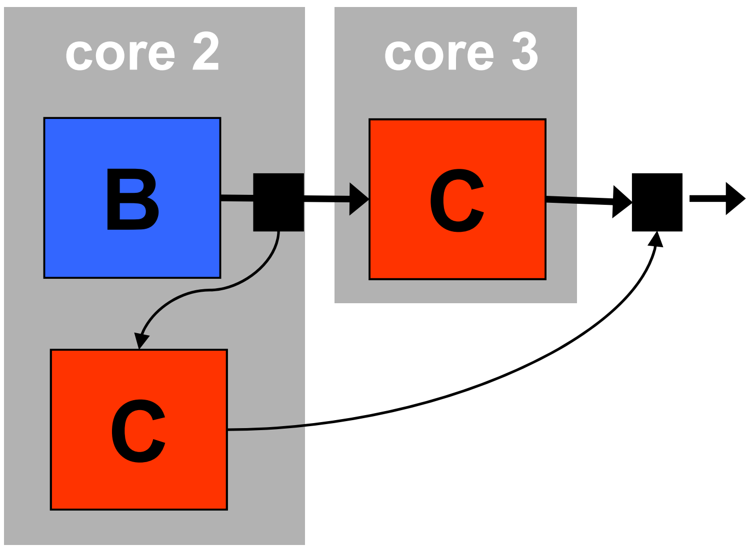
throughput = 2/5 = 0.4 < 0.429 (but > 0.333)

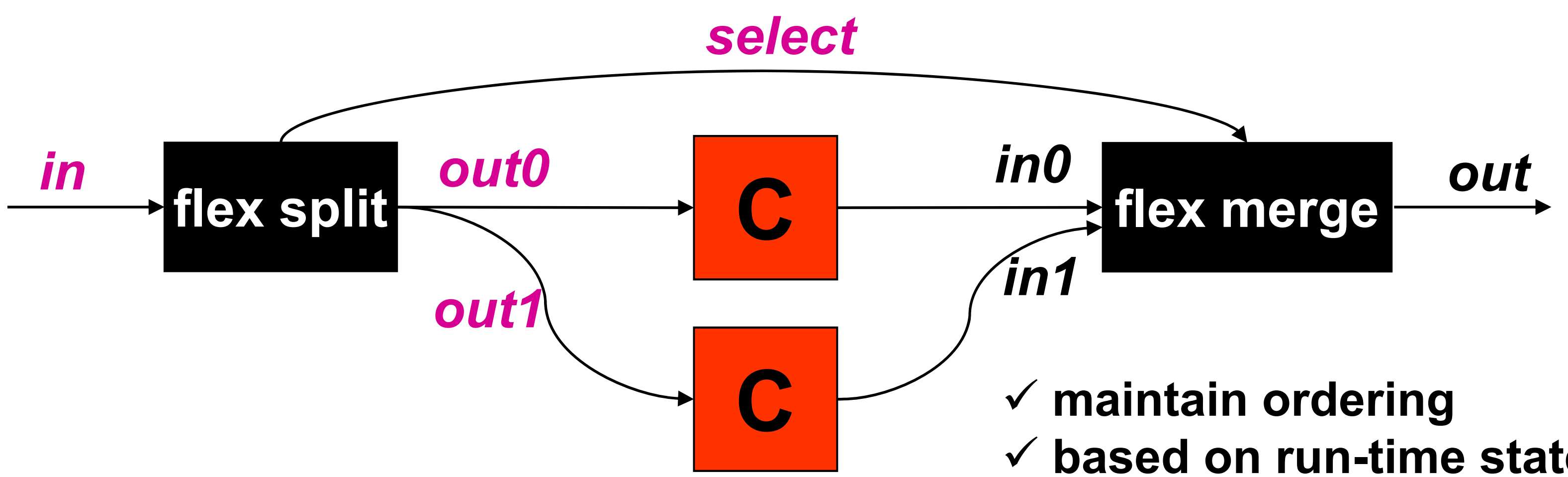
# Outline

- Introduction
- Design Flow of a Stream Program with Flexibility
- Performance
- Implementation of Flexible Filters
- Experiments
  - CFAR Case Study

# Flex-Split

```
Flex-split
   pop data block b from in
   n0 = available space on out0
   n1 = |b| - n0
   send n0 to out0, n1 to out1
   send n0 0's, then n1 1's to
       select
```

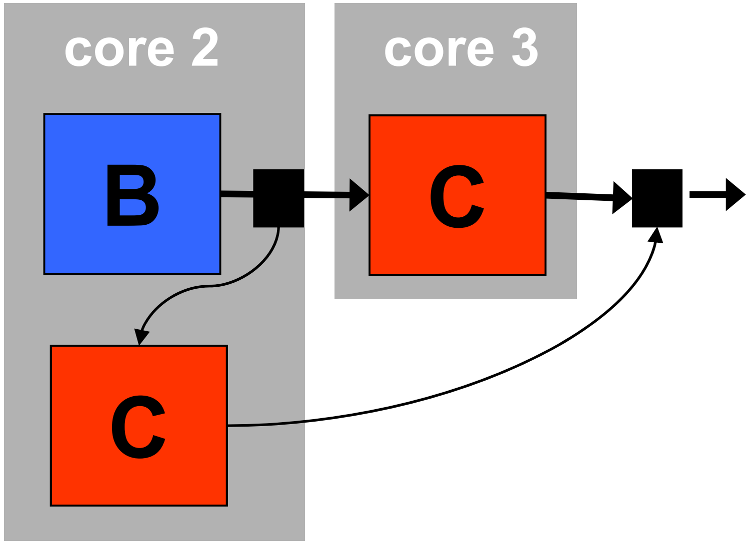


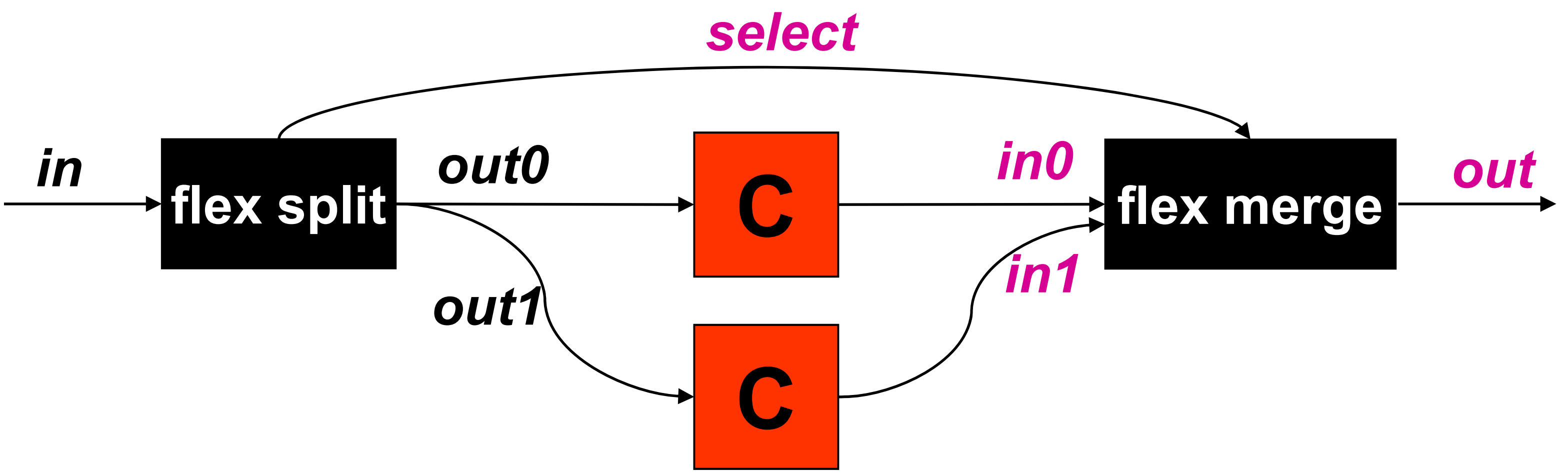




- ✓ maintain ordering
- ✓ based on run-time state of queues

# Flex-Merge

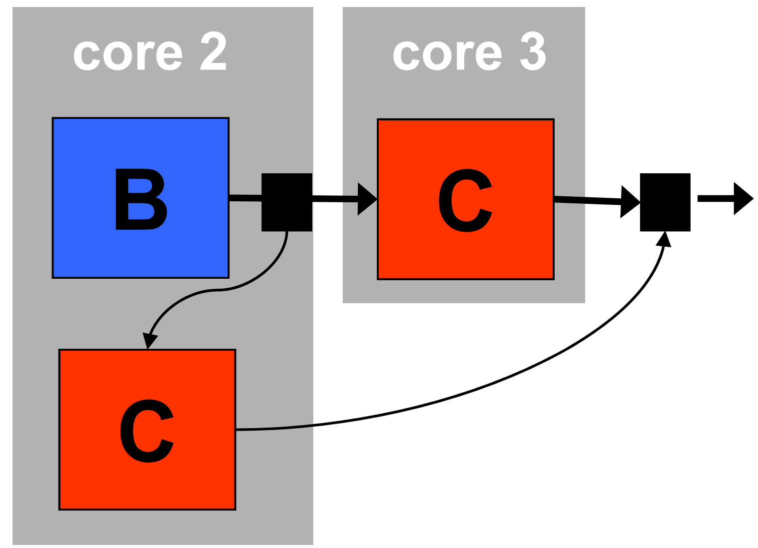
```
Flex-merge
   pop i from select
   if i is 0, pop token from in0
   if i is 1, pop token from in1
   push token to out
```

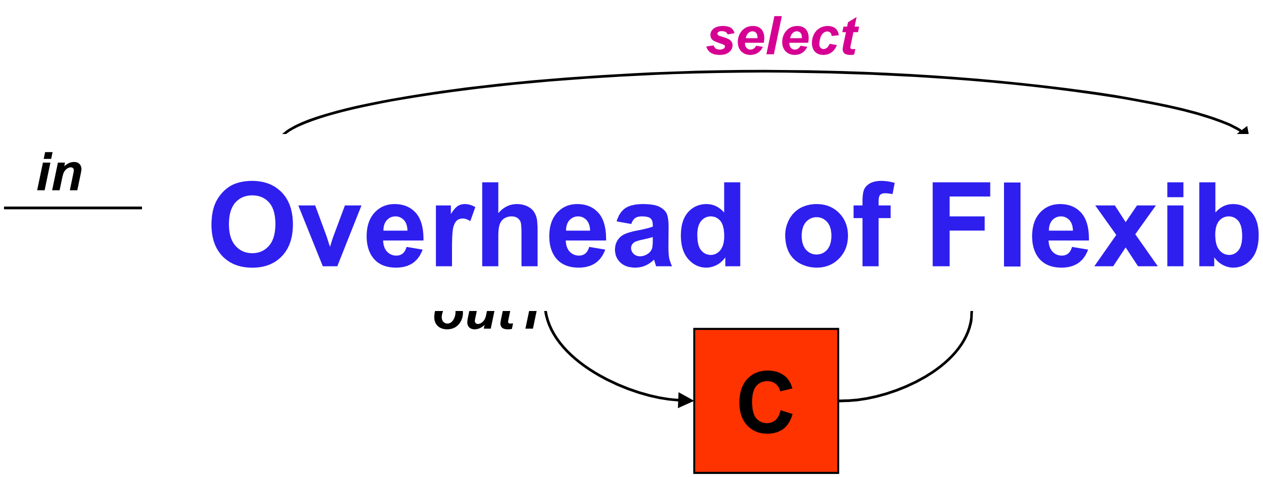


# Flex-Merge

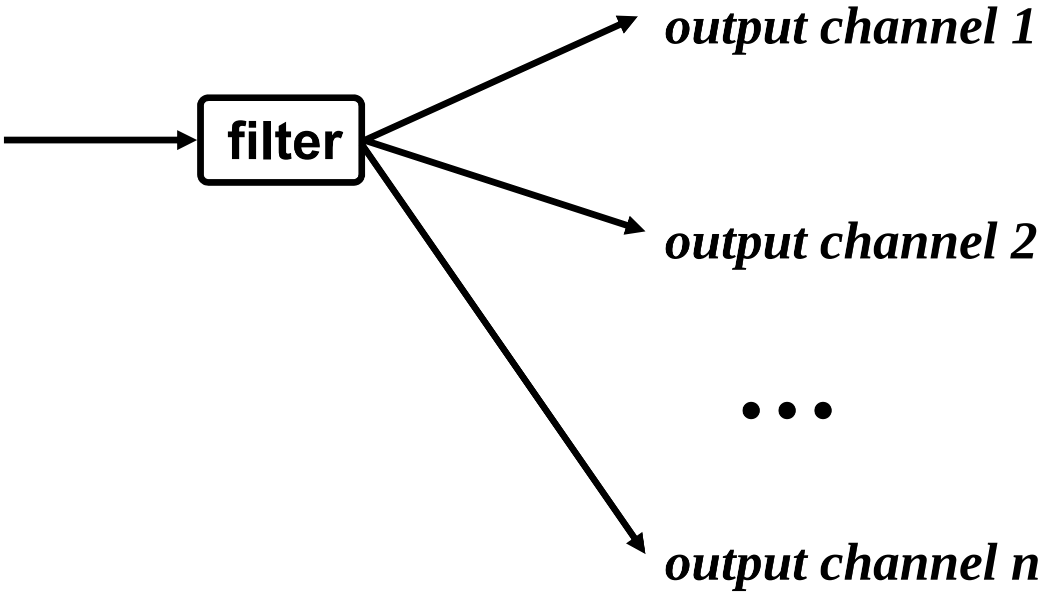
```
Flex-merge
   pop i from select
   if i is 0, pop token from in0
   if i is 1, pop token from in1
   push token to out
```



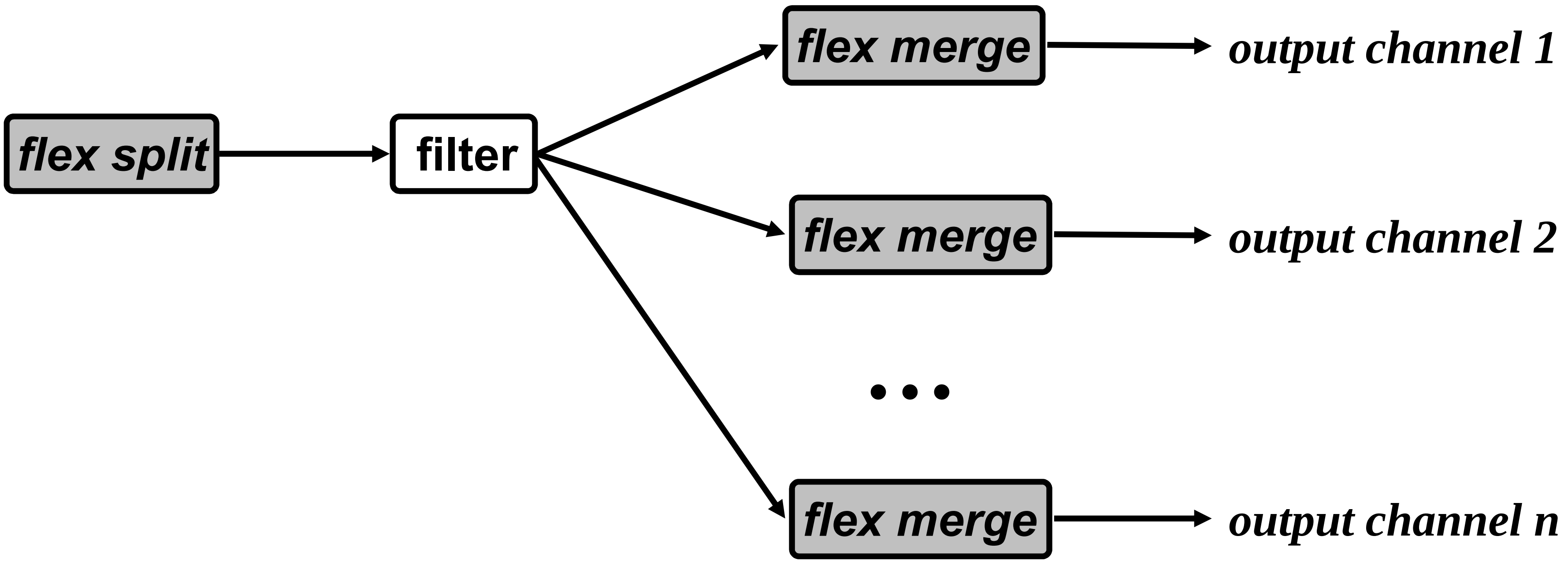




# Overhead of Flexibility?

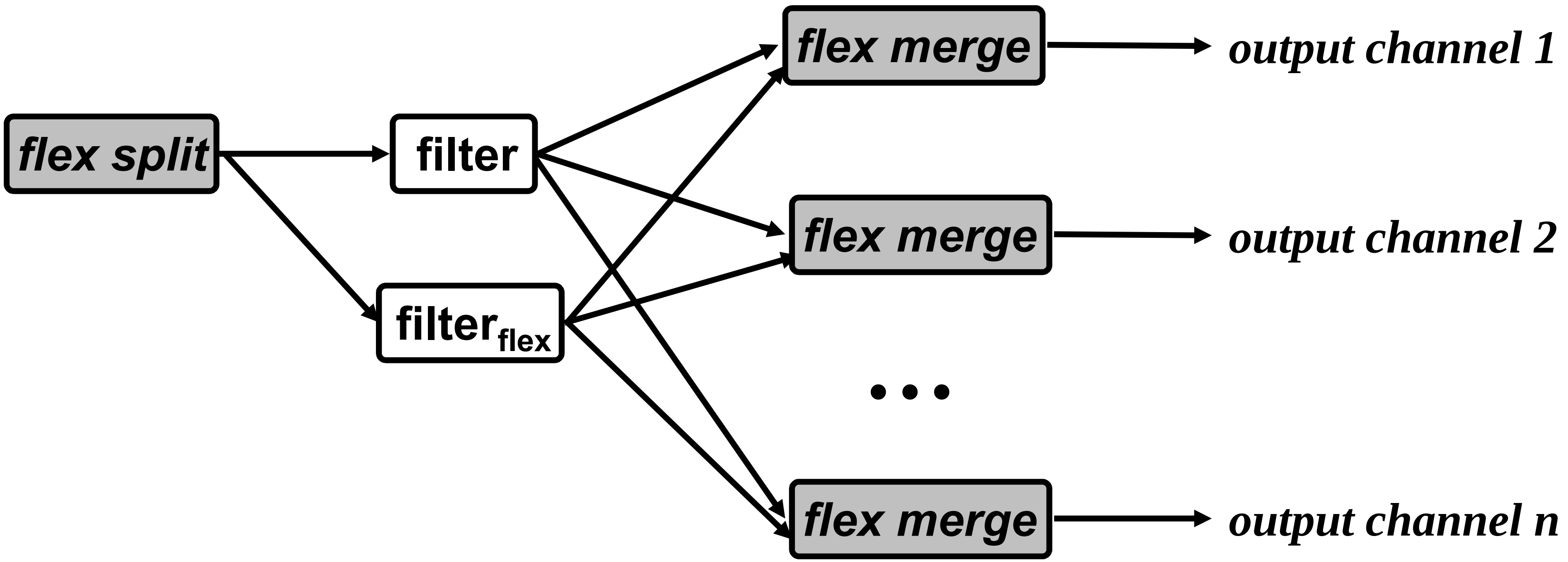
# Multi-Channel Flex-Split and Flex-Merge



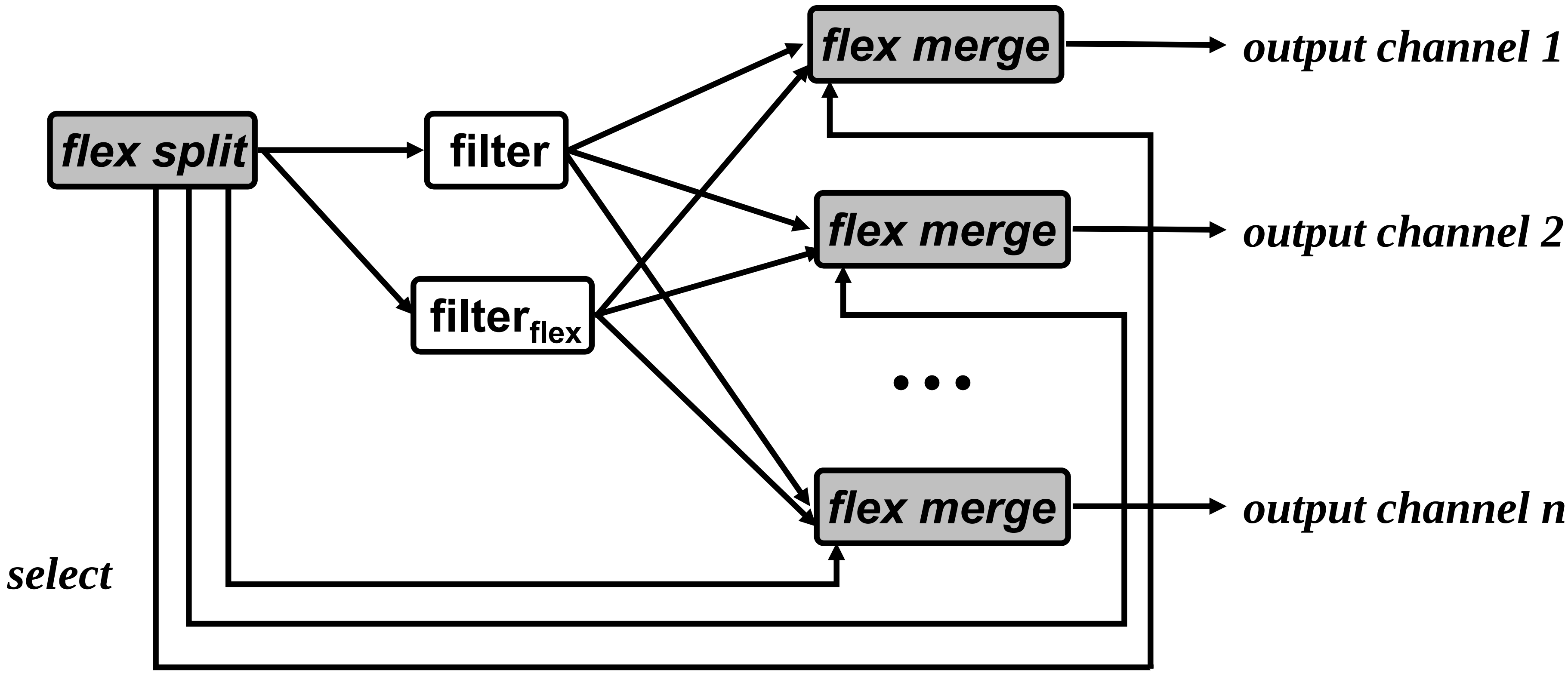
# Multi-Channel
# Flex-Split and Flex-Merge



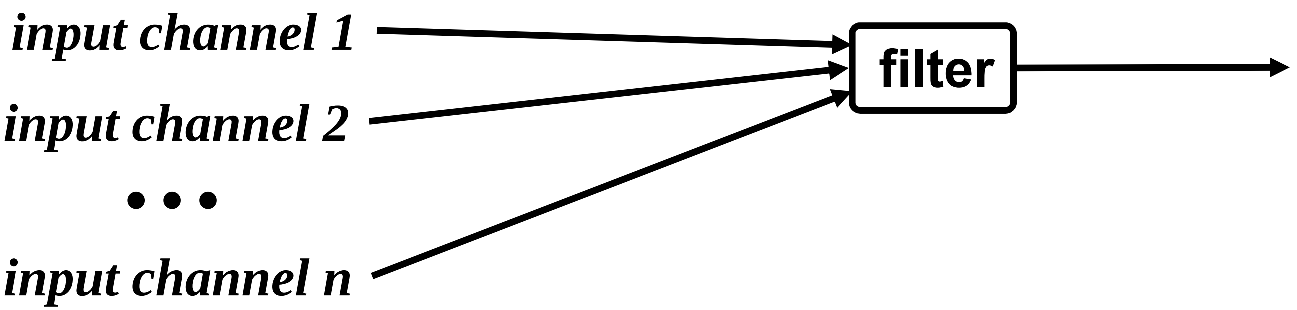
# Multi-Channel Flex-Split and Flex-Merge



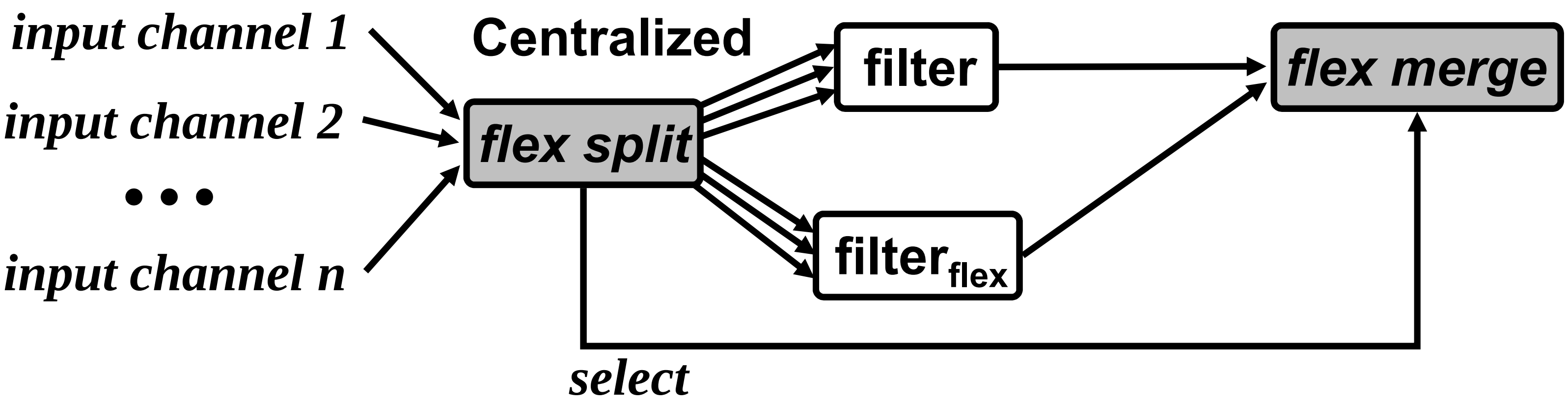
# Multi-Channel
# Flex-Split and Flex-Merge



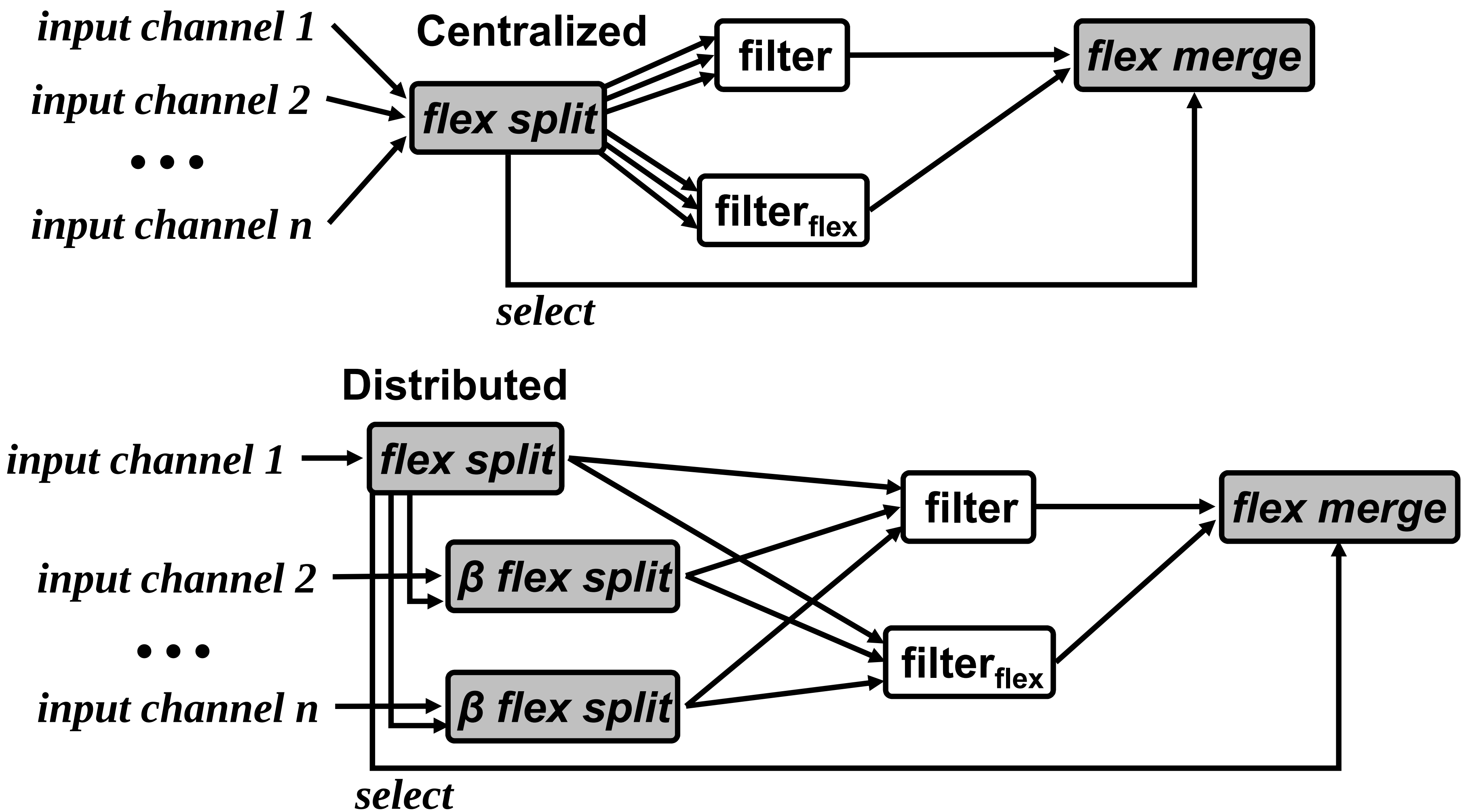
# Multi-Channel
# Flex-Split and Flex-Merge



# Multi-Channel Flex-Split and Flex-Merge



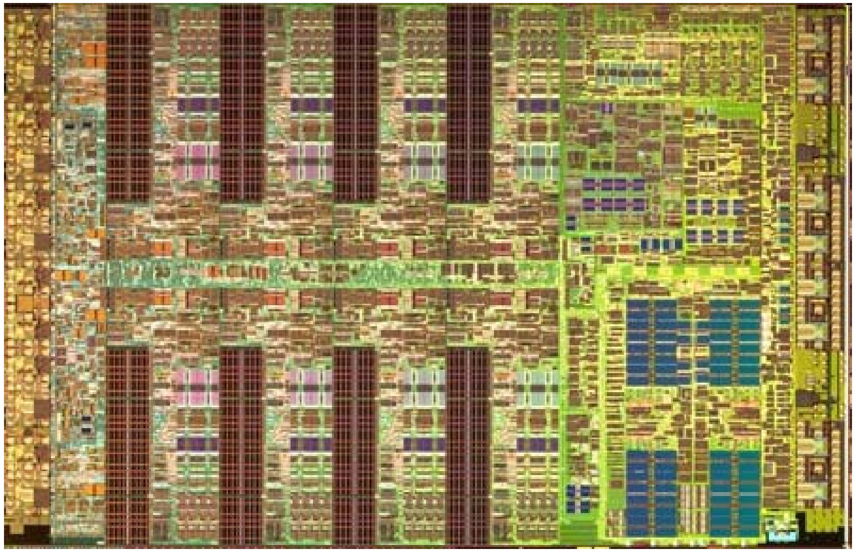
# Multi-Channel
# Flex-Split and Flex-Merge



# Outline

- Introduction
- Design Flow of a Stream Program with Flexibility
- Performance
- Implementation of Flexible Filters
- **Experiments**
  - **CFAR Case Study**

# Cell BE Processor

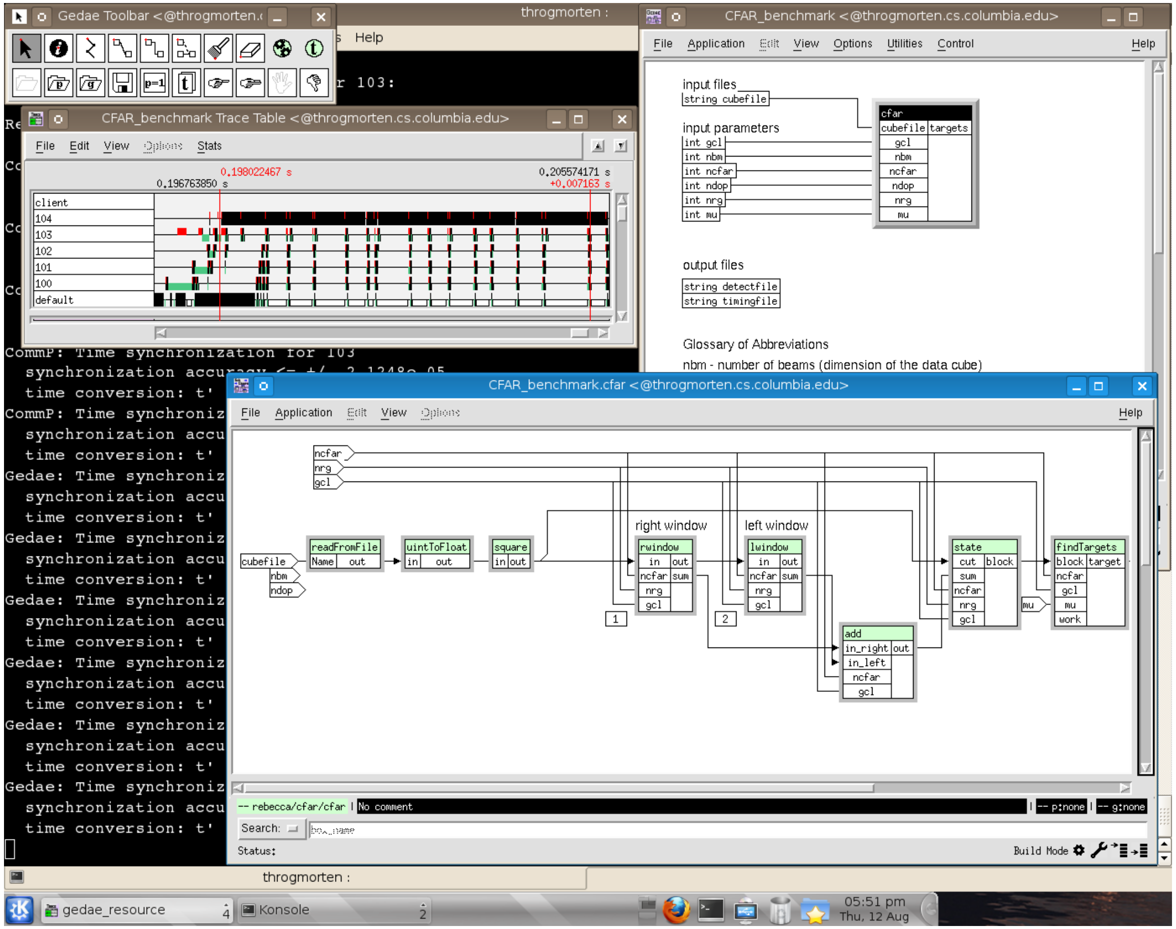


- Distributed Memory
- Heterogeneous
  - 8 SIMD (SPU) cores
  - 1 PowerPC (PPU)
- Element Interconnect Bus
  - 4 rings
  - 205 Gb/s
- Gedae Programming Language

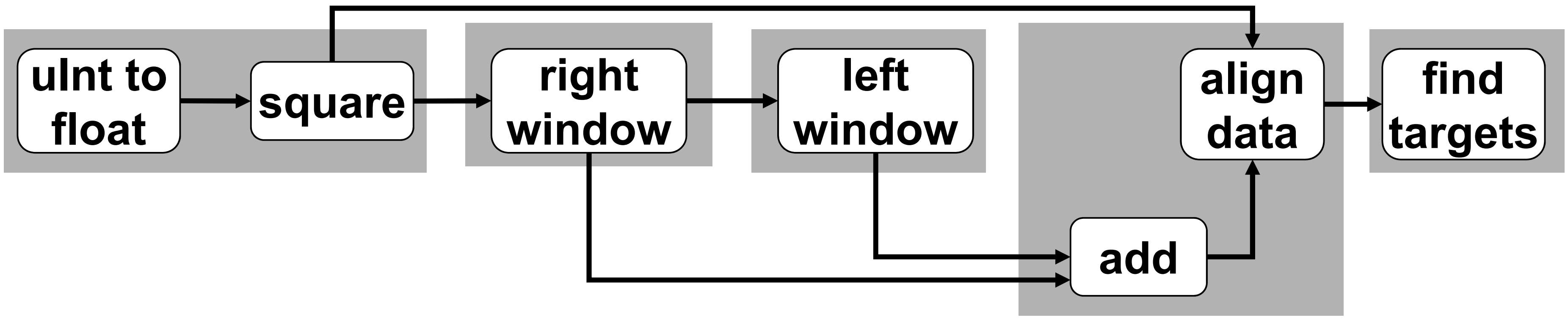
Communication Layer

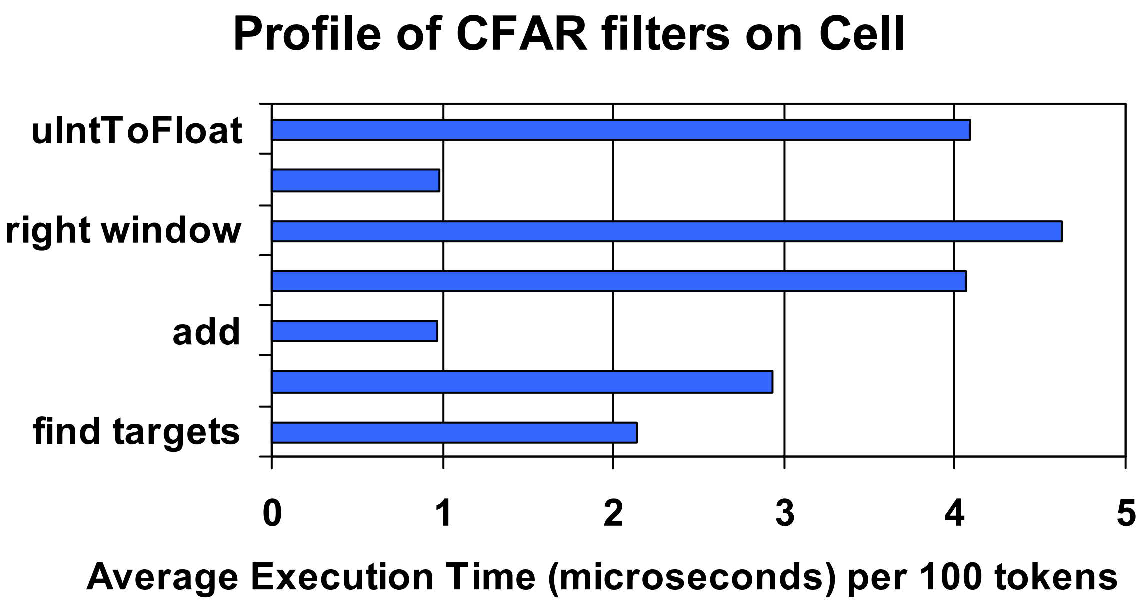
# Gedae

- Commercial data-flow language and programming GUI
- Performance analysis tools



# CFAR Benchmark



# Data Dependency

- By changing threshold, change % targets
  - 1.3 %
  - 7.3 %
- Additional workload per target
  - 16 µs
  - 32 µs
  - 64 µs

% Targets

| Additional Workload | | 1.3 | 7.3 |
|---|---|---|---|
| | 16 µs | 0.82 | 1.45 |
| | 32 µs | 1.06 | 1.39 |
| | 64 µs | 1.27 | 1.47 |

# More Benchmarks

| Benchmark | Field | Results | |
|---|---|---|---|
| Dedup | Information Theory | Rabin block/ max chunk size | Speedup |
| | | 4096/512 | 2.00 |
| JPEG | Image Processing | Image width x height | |
| | | 128x128<br>256x256<br>512x512 | 1.31<br>1.16<br>1.25 |
| Value-at-Risk | Finance | stocks/walks/timesteps | |
| | | 16/1024/1024<br>64/1024/1024<br>128/1024/1024 | 0.98<br>1.56<br>1.55 |

# Conclusions

- Flexible filters
  - adapt to data dependent bottlenecks
  - distributed load balancing
  - provide speedup without modification to original filters
  - can be implemented on top of general stream languages