Programming Practice

Fall 2011

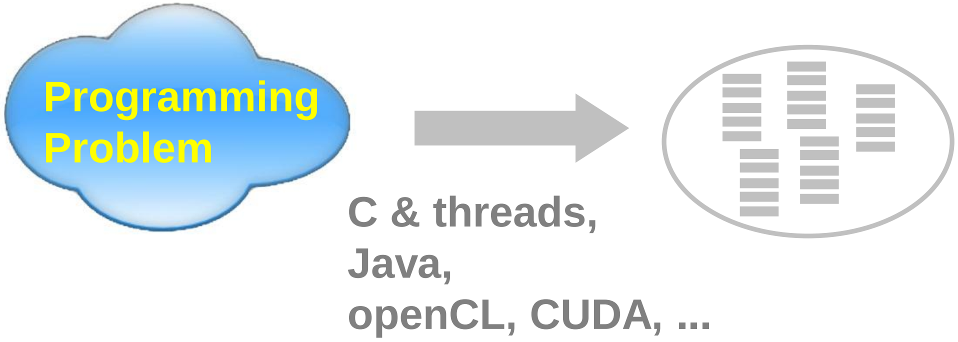
Stream-Parallelism

StreamIt

Bernd Burgstaller
Yonsei University

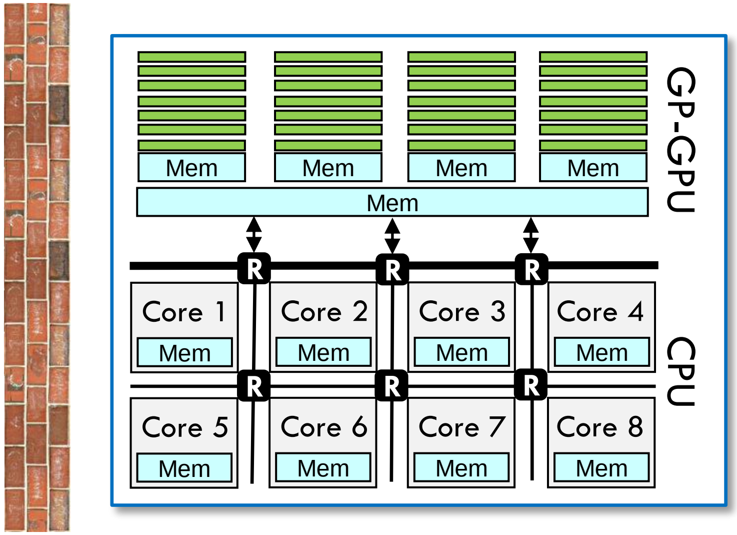
Bernhard Scholz
The University of Sydney

Today: Programmers are challenged!



Programming Problem

C & threads,
Java,
openCL, CUDA, ...



Current programming methods support multicore architectures **poorly**:

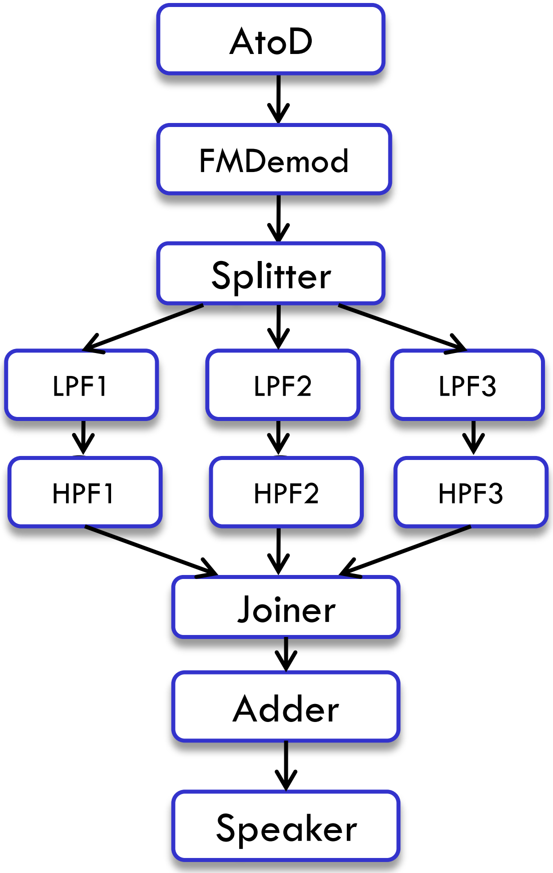
- □ Not enough parallelism
- □ Concurrency bugs
- □ Performance bugs
- □ Wrong abstraction levels:
  - Too low to be productive
  - Mostly too high for performance
  - Parallel hw exposed to programmer

- □ The parallel programming gap:
  - between capabilities of today's compilers and programming language implementations, and the complexity of parallel architectures and applications.

New programming abstractions required!

Stream Programming:

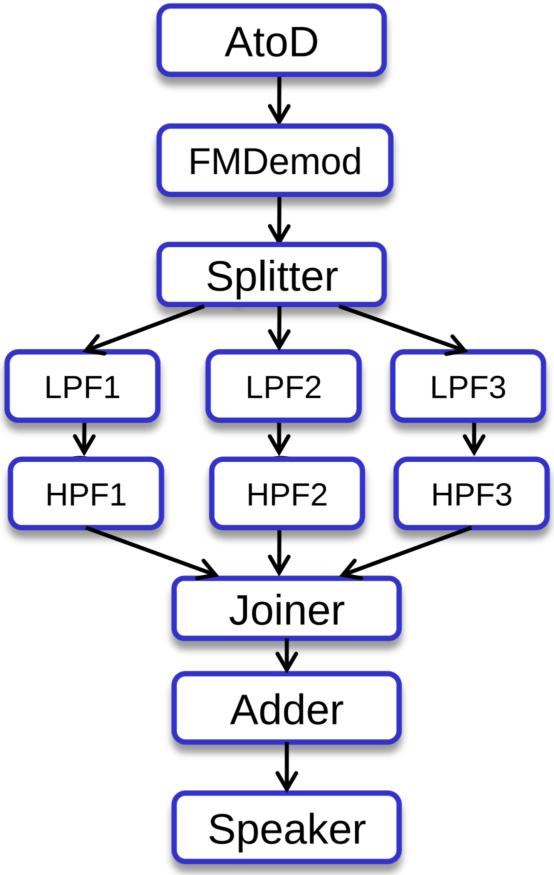
- For programs based on regular streams of data
  - Audio, video, DSP, networking, cryptography, graphics
  - Languages: StreamIt, Brook, Baker, StreamFlex, Cg, …
- Intuitive programing abstraction:
  - Streams
    - FIFO data channels
  - Actors
    - Basic unit of computation
    - Independent tasks!
    - Communication restricted to input/output stream(s)
  - Task, data and pipeline parallelism ☺
- Amenable to aggressive compiler optimizations
  - [ASPLOS'02, '06, PLDI '03, '08, ASPLOS'11]
- High programmer productivity
  - abstracts away from underlying heterogeneous multicore hardware!



Streaming models of computation

- Many different ways to represent stream programs:
  - Do senders/receivers block?
  - How much buffering is allowed on channels?
  - Is computation deterministic?
  - Can you avoid deadlock?

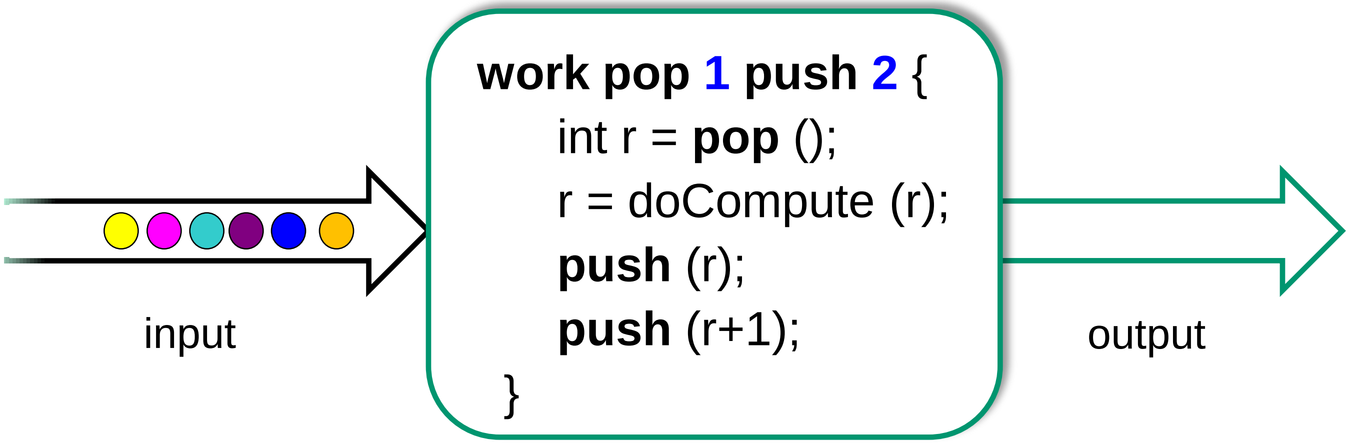
- Three common models:
  1) Kahn Process Networks
  2) Synchronous Dataflow
  3) Communicating Sequential Processes



The StreamIt Language

- Developed at MIT
  - started at around the year 2000
- A high-level, architecture-independent language for streaming applications
  - Improves multicore programmer productivity (unlike Java, C)
  - Offers scalable performance on multicores
- Based on synchronous dataflow, with dynamic extensions
  - Compiler determines execution order of filters
  - Many aggressive optimizations possible

Filters as basic units of computation



```
work pop 1 push 2 {
      int r = pop ();
      r = doCompute (r);
      push (r);
      push (r+1);
  }
```

With each iteration of this filter's work function:

1.) one data element is popped from input channel

2.) computation is performed,

3.) two result-values are pushed onto output channel

peek(n) operation:

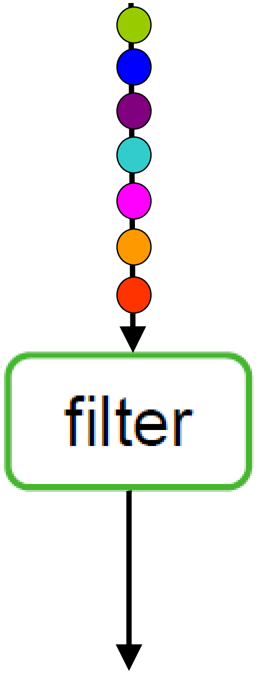
read nth value from input channel without consuming

Filter Overview

- Filters are the basic unit of computation
- Filters communicate with neighboring filters using typed FIFO channels
- Channels support three operations:
  - **`pop()`**: remove item from end of input channel
  - **`peek(i)`**: read value i slots up the input channel (non-consuming)
  - **`push(value)`**: push value onto filter's output channel

- Each filter contains:
  - an **`init(…)`** function called an initialization time.
  - a **`work()`** function to describe one execution of the filter
  - possible helper functions called by **`init()`** or **`work()`**
  - variables persistent over executions of the work() function
    - called the state of a filter

Filters are the basic unit of computation

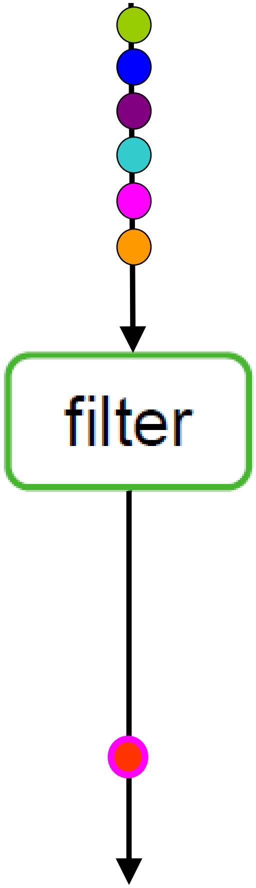


```
work pop 1 push 1 {
  float r = pop();
  r = compute_something(r);
  push (r);
}
```

With each iteration of the filter’s work function,

1) one data element is popped from the stream,
2) a computation is performed,
3) the resulting value is pushed onto the stream.

Filters are the basic unit of computation (cont.)

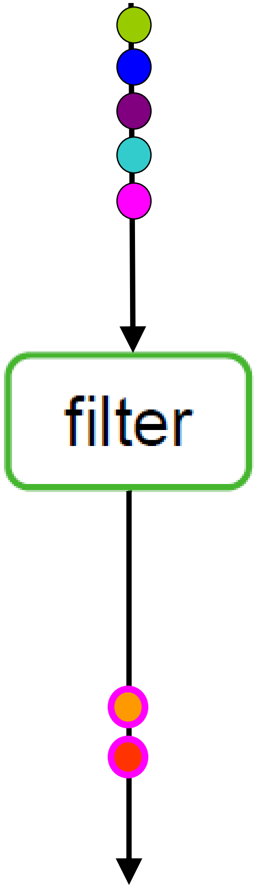


```
work pop 1 push 1 {
  float r = pop();
  r = compute_something(r);
  push (r);
}
```

With each iteration of the filter's work function,

1) one data element is popped from the stream,
2) a computation is performed,
3) the resulting value is pushed onto the stream.

Filters are the basic unit of computation (cont.)

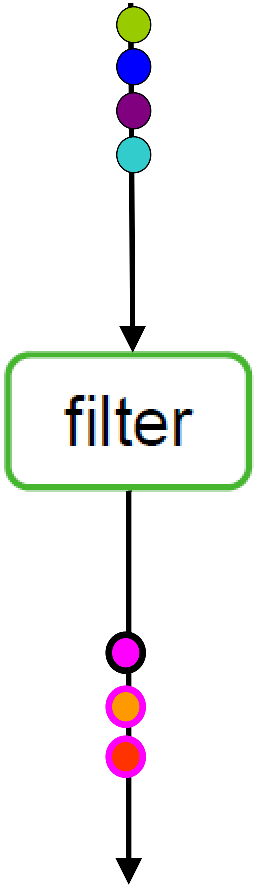


```
work pop 1 push 1 {
  float r = pop();
  r = compute_something(r);
  push (r);
}
```

With each iteration of the filter’s work function,

1) one data element is popped from the stream,
2) a computation is performed,
3) the resulting value is pushed onto the stream.

Filters are the basic unit of computation (cont.)

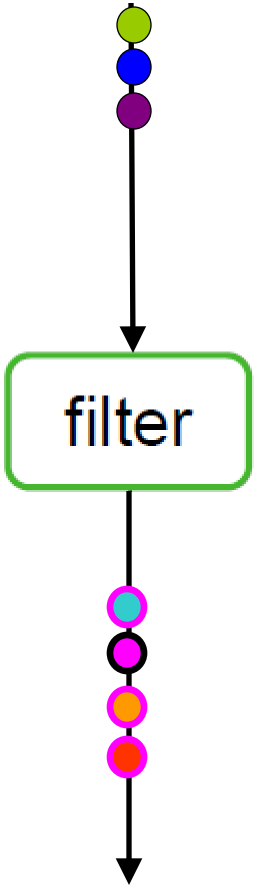


```
work pop 1 push 1 {
  float r = pop();
  r = compute_something(r);
  push (r);
}
```

With each iteration of the filter’s work function,

1) one data element is popped from the stream,
2) a computation is performed,
3) the resulting value is pushed onto the stream.

Filters are the basic unit of computation (cont.)

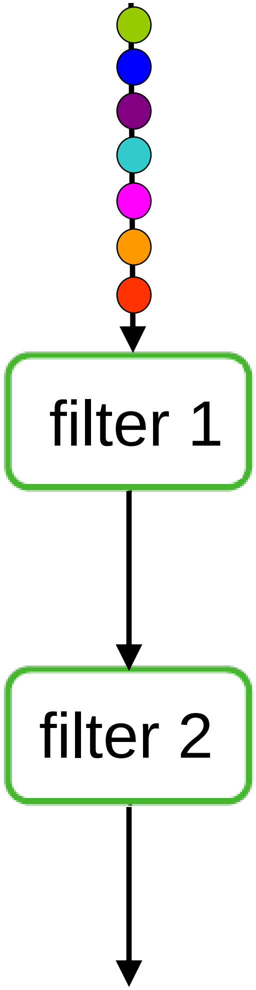


```
work pop 1 push 1 {
  float r = pop();
  r = compute_something(r);
  push (r);
}
```

With each iteration of the filter's work function,

1) one data element is popped from the stream,
2) a computation is performed,
3) the resulting value is pushed onto the stream.

Example pipeline: a sequence of 2 filters



filter 1

```
work pop 1 push 1 {
  float r = pop();
  r = compute_something (r);
  push (r);
}
```

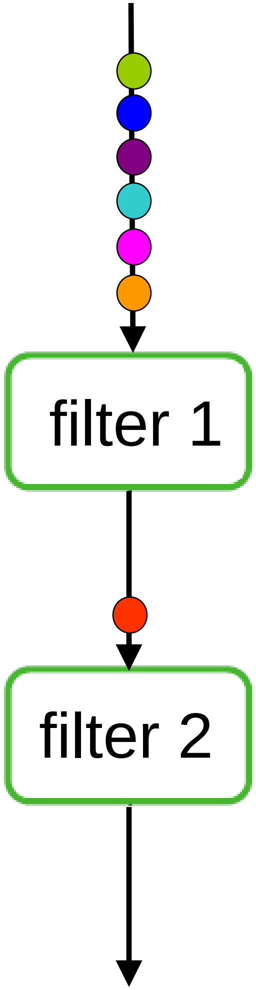
filter 2

```
work pop 1 push 1 {
  float r = pop();
  r = compute_somethingelse (r);
  push (r);
}
```

Two filters, one workfunction each:

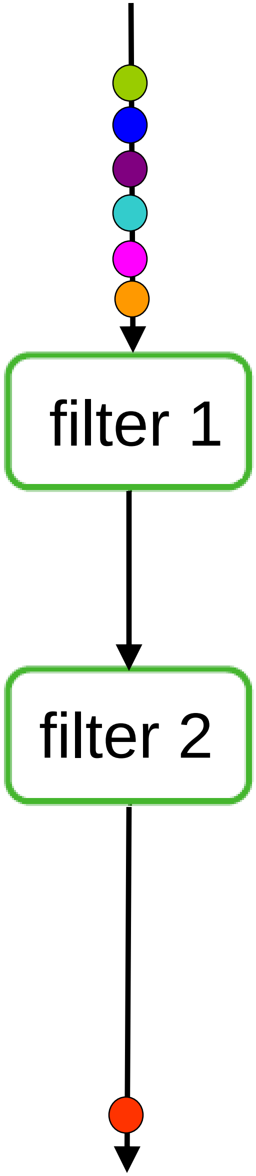
- Output of filter 1 becomes input of filter 2
- A filter buffers its input until it has received at least as many items as it pops, then it “fires”
- the computed value(s) is/are pushed onto the output stream.

Example pipeline: a sequence of 2 filters



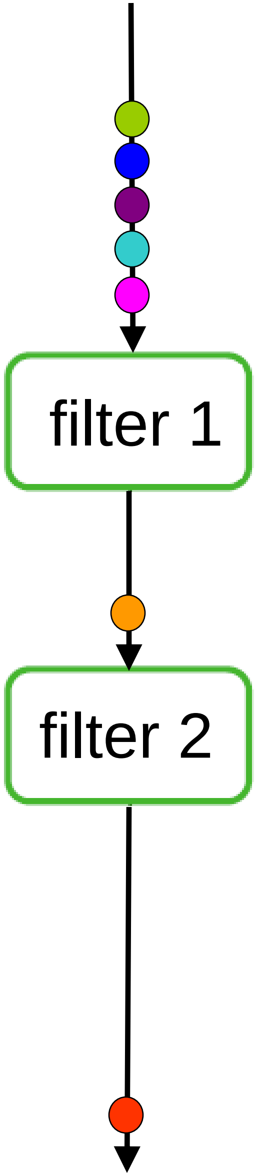
- Filter 2 can fire only after filter 1 has produced the first data item.

Example pipeline: a sequence of 2 filters



- Filter 2 can fire only after filter 1 has produced the first data item.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.

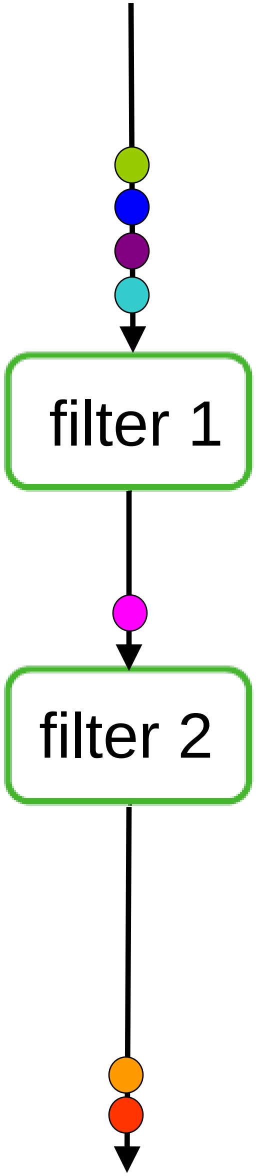
Example pipeline: a sequence of 2 filters



- Filter 2 can fire only after filter 1 has produced the first data item.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.
- On a single CPU, we can schedule Filter 1 and Filter 2 as follows (this is done automatically, not by the programmer!):

```
while(1) {
    work_filter1();
    work_filter2();
}
```

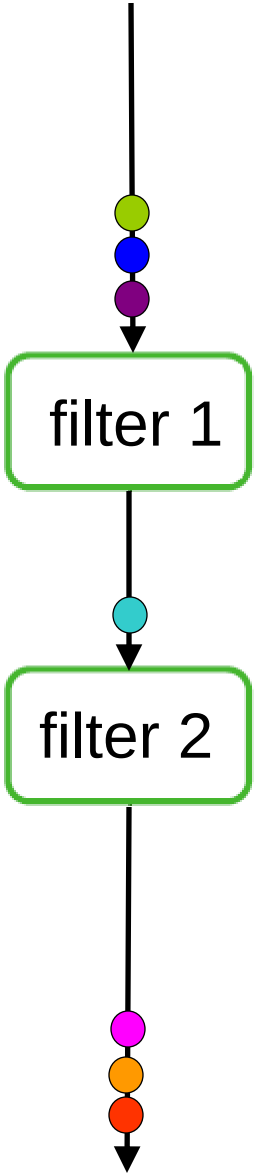
Example pipeline: a sequence of 2 filters



- Filter 2 can fire only after filter 1 has produced the first data item.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.
- On a single CPU, we can schedule Filter 1 and Filter 2 as follows (this is done automatically, not by the programmer!):

```
while(1) {
    work_filter1();
    work_filter2();
}
```

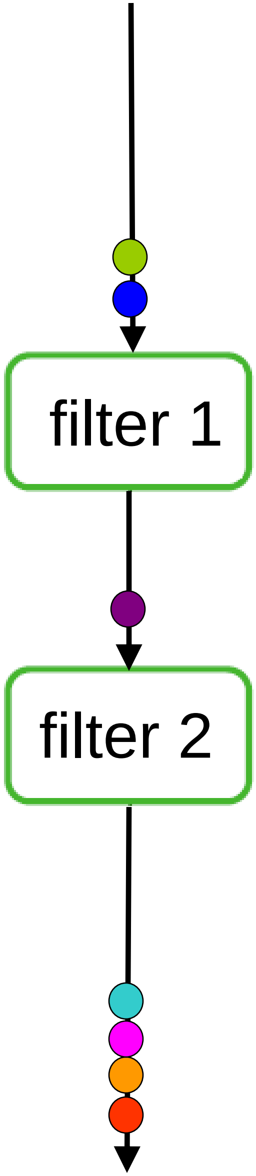
Example pipeline: a sequence of 2 filters



- Filter 2 can fire only after filter 1 has produced the first data item.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.
- On a single CPU, we can schedule Filter 1 and Filter 2 as follows (this is done automatically, not by the programmer!):

```
while(1) {
    work_filter1();
    work_filter2();
}
```

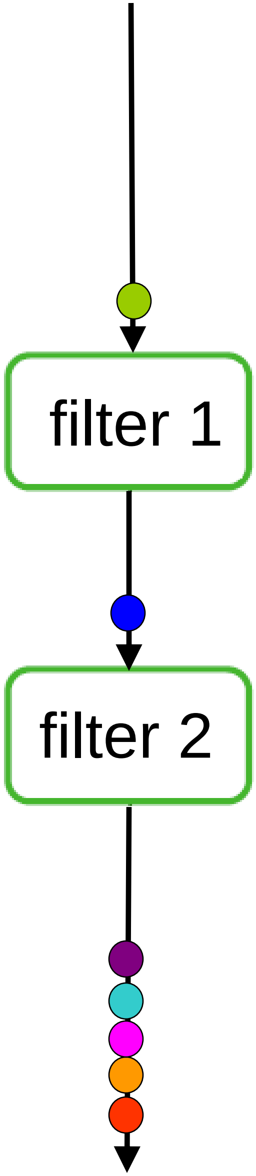
Example pipeline: a sequence of 2 filters



- Filter 2 can fire only after filter 1 has produced the first data item.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.
- On a single CPU, we can schedule Filter 1 and Filter 2 as follows (this is done automatically, not by the programmer!):

```
while(1) {
    work_filter1();
    work_filter2();
}
```

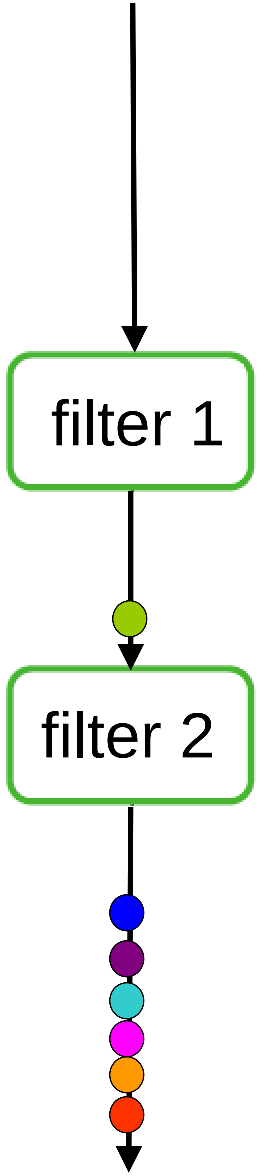
Example pipeline: a sequence of 2 filters



- Filter 2 can fire only after filter 1 has produced the first data item.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.
- On a single CPU, we can schedule Filter 1 and Filter 2 as follows (this is done automatically, not by the programmer!):

```
while(1) {
    work_filter1();
    work_filter2();
}
```

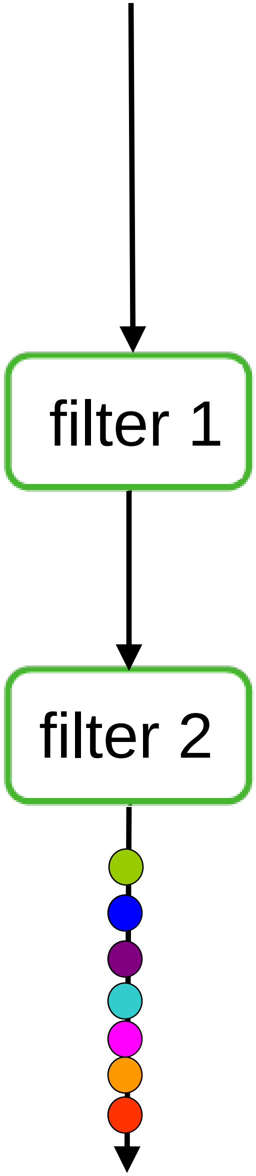
Example pipeline: a sequence of 2 filters



- Filter 2 can fire only after filter 1 has produced the first data item.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.
- On a single CPU, we can schedule Filter 1 and Filter 2 as follows (this is done automatically, not by the programmer!):

```
while(1) {
    work_filter1();
    work_filter2();
}
```

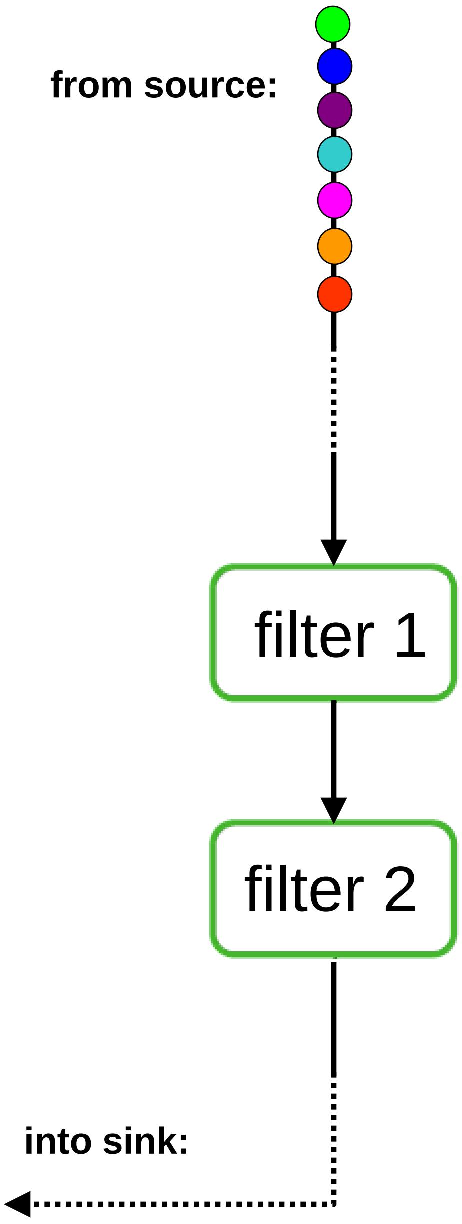
Example pipeline: a sequence of 2 filters



- Filter 2 can fire only after filter 1 has produced the first data item.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.
- On a single CPU, we can schedule Filter 1 and Filter 2 as follows (this is done automatically, not by the programmer!):

```
while(1) {
    work_filter1();
    work_filter2();
}
```

Example 2: two filters with different input data rates



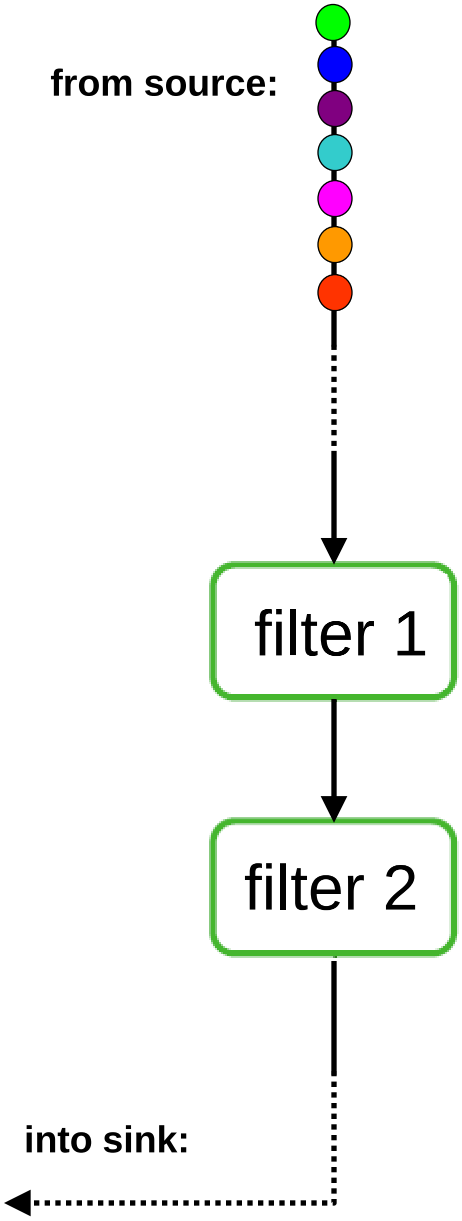
filter 1

```
work pop 1 push 1 {
  float r = pop();
  r = compute_something(r);
  push (r);
}
```

filter 2

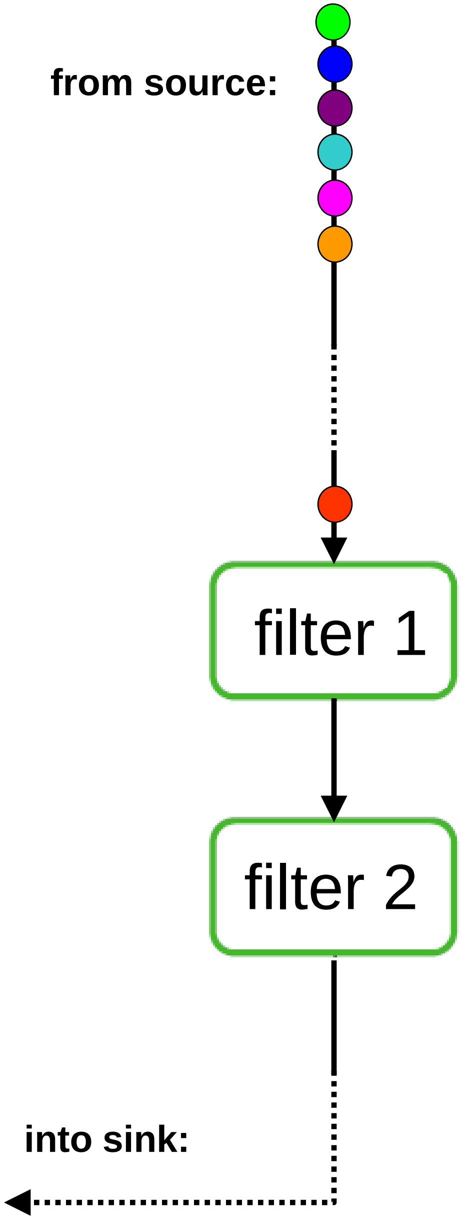
```
work pop 2 push 1 {
  float r = pop();
  float r1 = pop();
  r = compute_something_else(r, r1);
  push (r);
}
```

Example 2: two filters with different input data rates



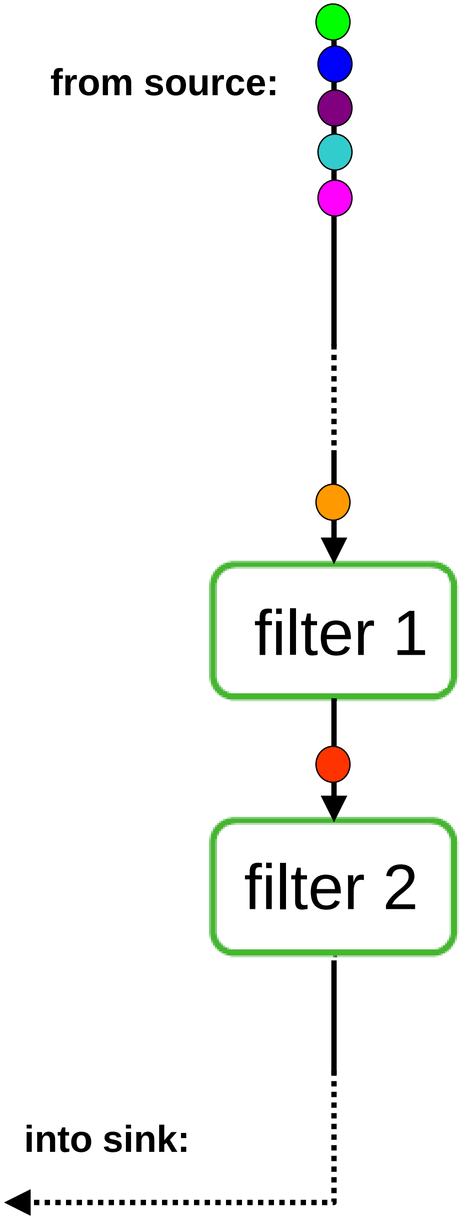
- Filter 2 can fire only after filter 1 has produced at least 2 data items.
  - Filter 1 needs to execute two times for each execution of Filter2.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.

Example 2: two filters with different input data rates



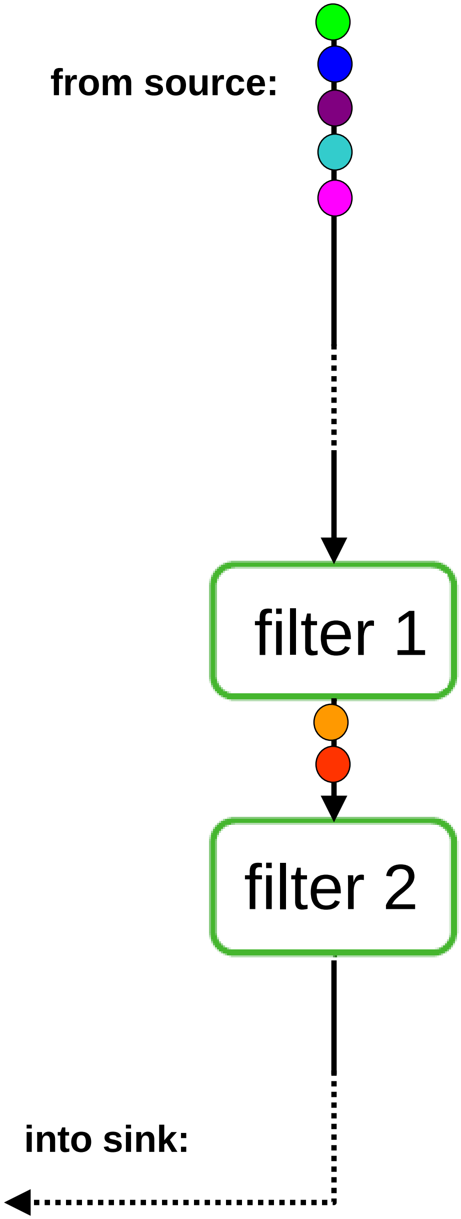
- Filter 2 can fire only after filter 1 has produced at least 2 data items.
  - Filter 1 needs to execute two times for each execution of Filter2.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.

Example 2: two filters with different input data rates



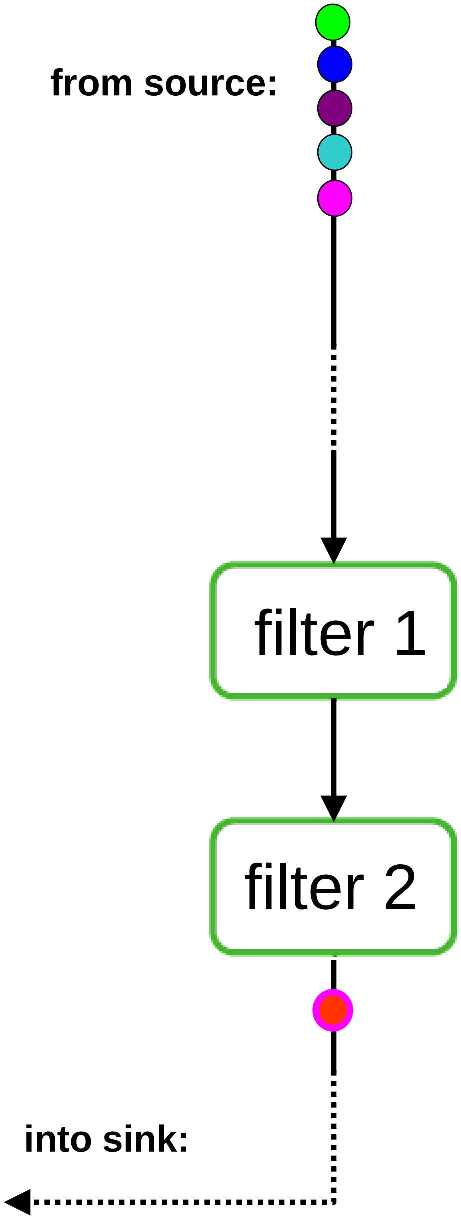
- Filter 2 can fire only after filter 1 has produced at least 2 data items.
  - Filter 1 needs to execute two times for each execution of Filter2.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.

Example 2: two filters with different input data rates



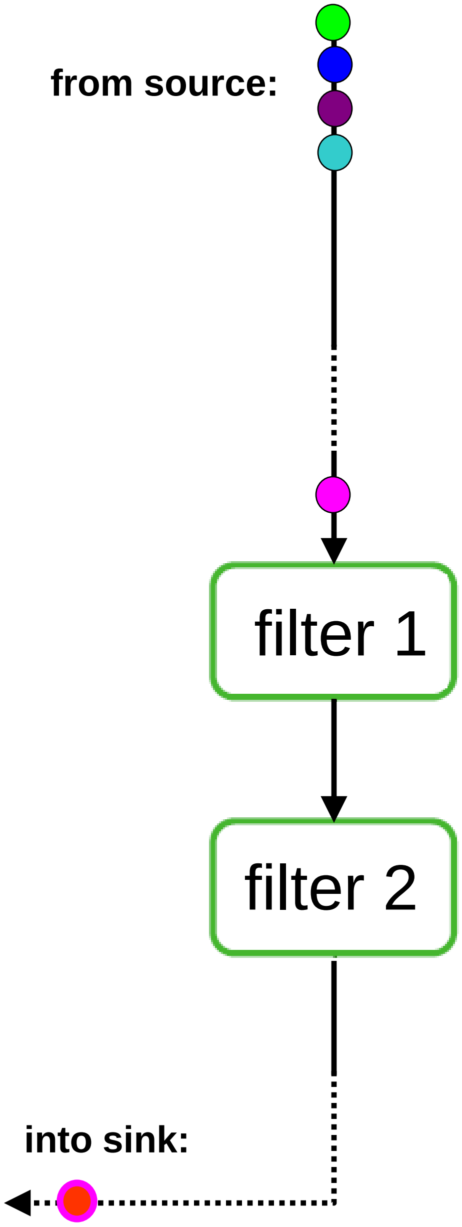
- Filter 2 can fire only after filter 1 has produced at least 2 data items.
    - Filter 1 needs to execute two times for each execution of Filter2.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.

Example 2: two filters with different input data rates



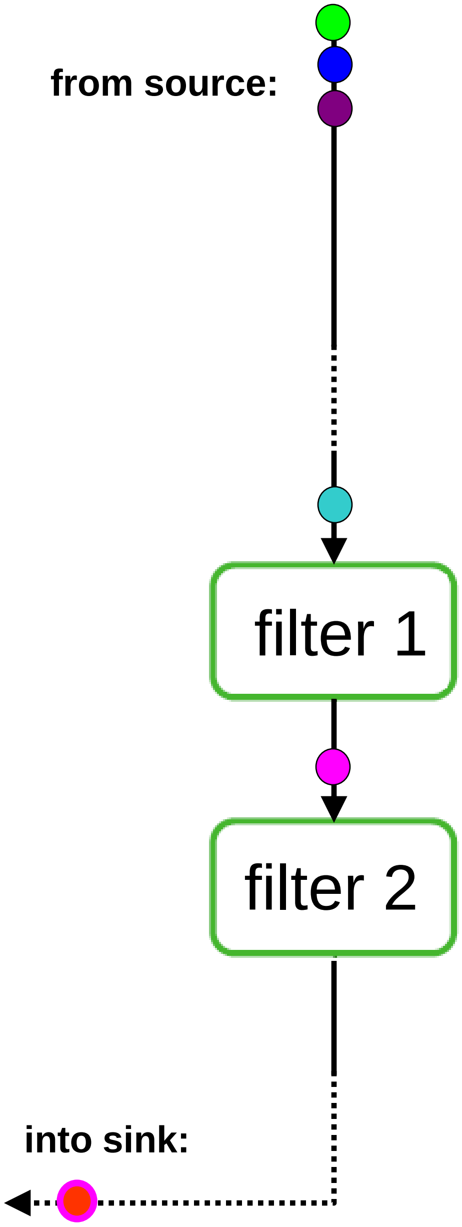
- Filter 2 can fire only after filter 1 has produced at least 2 data items.
  - Filter 1 needs to execute two times for each execution of Filter2.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.

Example 2: two filters with different input data rates



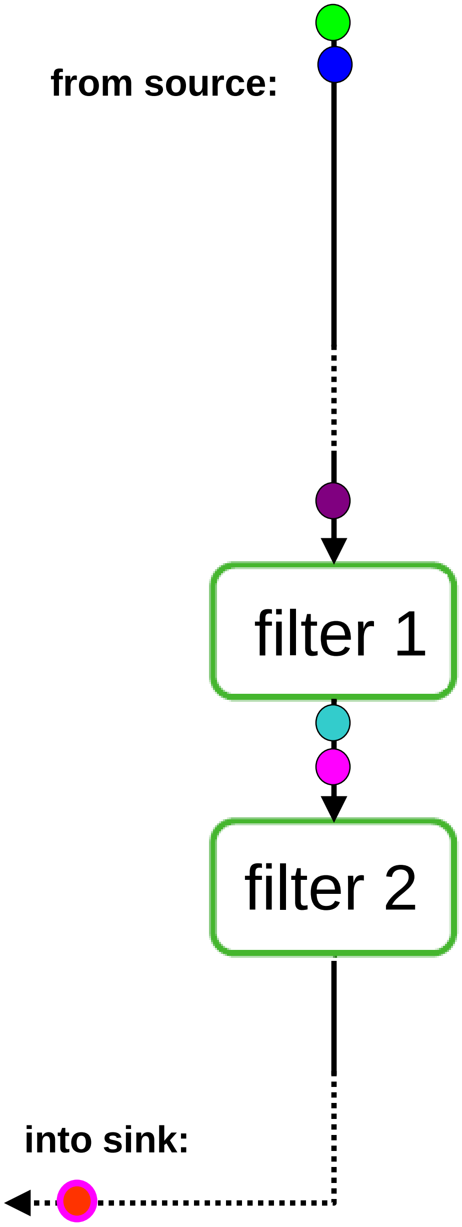
- Filter 2 can fire only after filter 1 has produced at least 2 data items.
  - Filter 1 needs to execute two times for each execution of Filter2.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.

Example 2: two filters with different input data rates



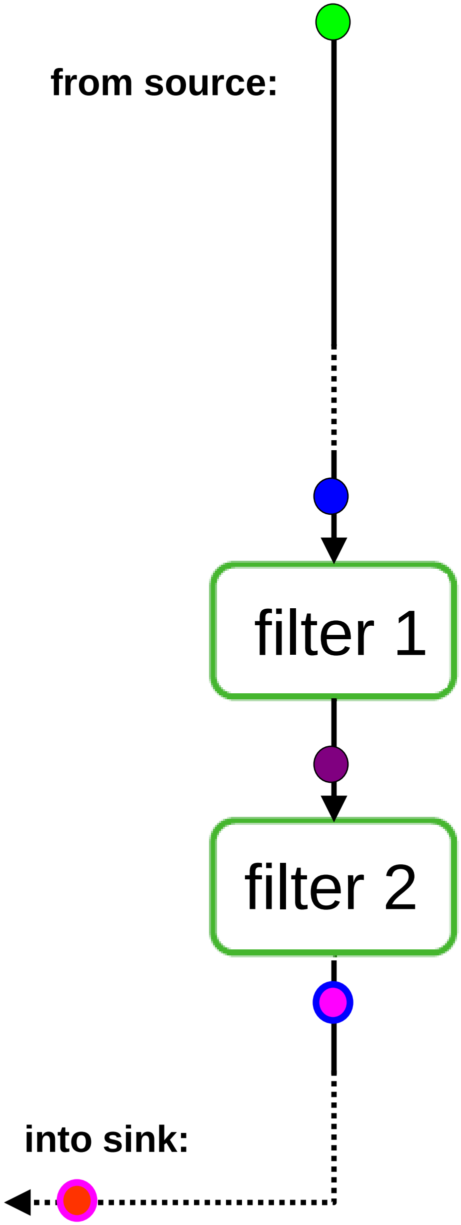
- Filter 2 can fire only after filter 1 has produced at least 2 data items.
    - Filter 1 needs to execute two times for each execution of Filter2.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.

Example 2: two filters with different input data rates



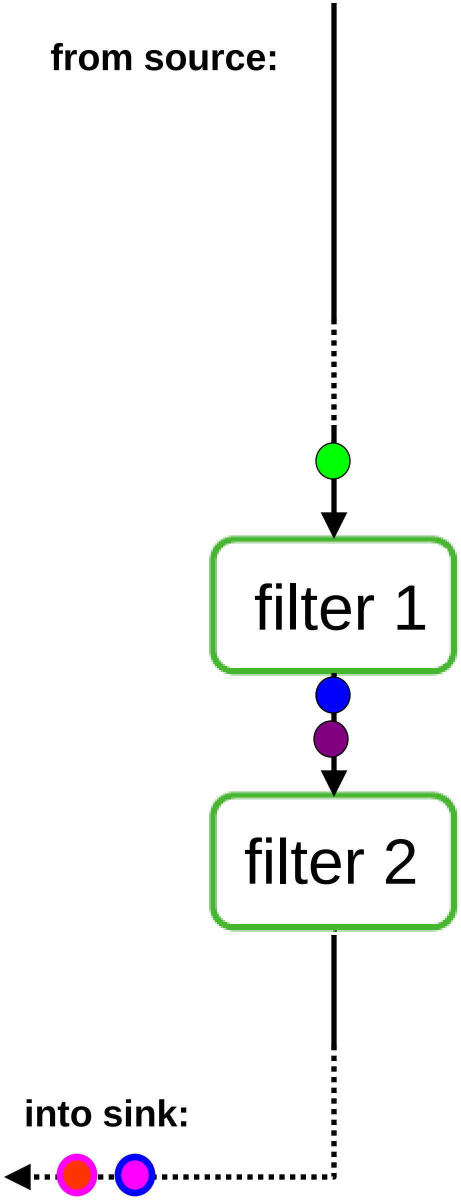
- Filter 2 can fire only after filter 1 has produced at least 2 data items.
  - Filter 1 needs to execute two times for each execution of Filter2.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.

Example 2: two filters with different input data rates



- Filter 2 can fire only after filter 1 has produced at least 2 data items.
  - Filter 1 needs to execute two times for each execution of Filter2.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.

Example 2: two filters with different input data rates



from source:

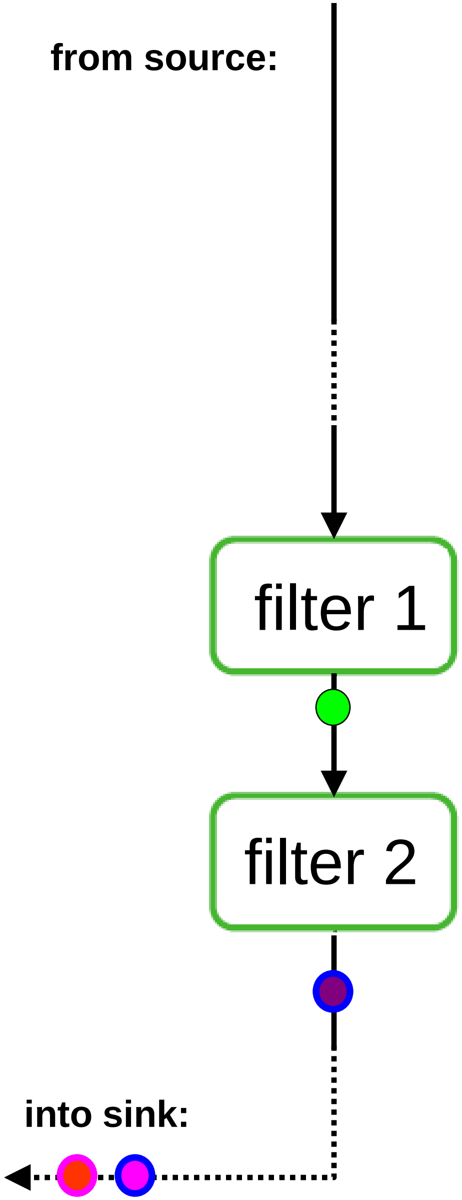
filter 1

filter 2

into sink:

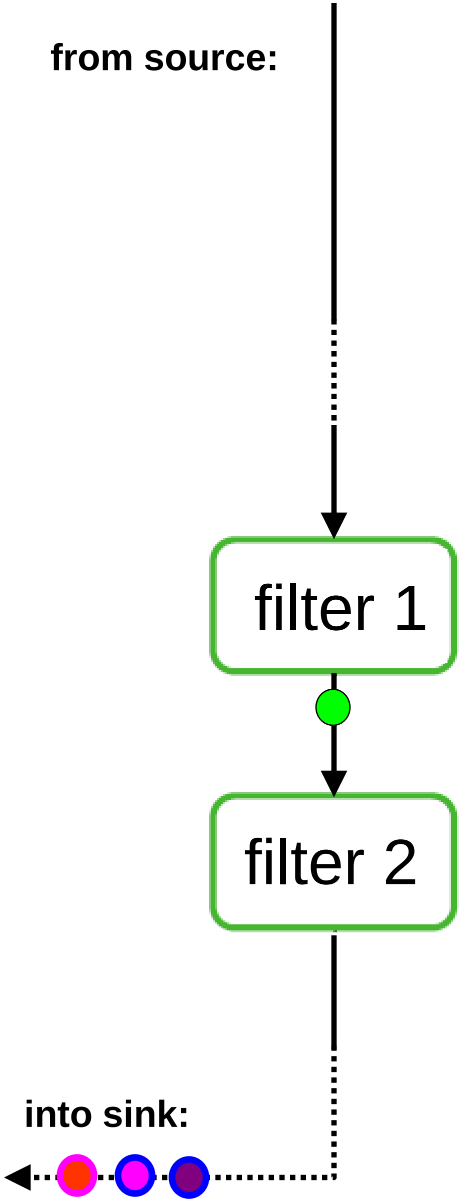
- Filter 2 can fire only after filter 1 has produced at least 2 data items.
  - Filter 1 needs to execute two times for each execution of Filter2.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.

Example 2: two filters with different input data rates



- Filter 2 can fire only after filter 1 has produced at least 2 data items.
    - Filter 1 needs to execute two times for each execution of Filter2.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.

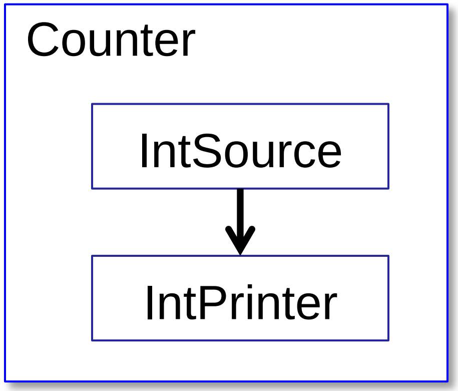
Example 2: two filters with different input data rates



- Filter 2 can fire only after filter 1 has produced at least 2 data items.
  - Filter 1 needs to execute two times for each execution of Filter2.
- If we run Filter 1 and Filter 2 on different CPUs/cores, they can execute in parallel.

StreamIt example: a simple counter

```
void→void pipeline Counter {
   add IntSource ();
   add IntPrinter ();
}

void→int filter IntSource () {
   int ctr;
   init { ctr = 0; }
   work push 1 {
      push (ctr++);
   }

int→void filter IntPrinter () {
   work pop 1 { print(pop()); }
}
```



Counter

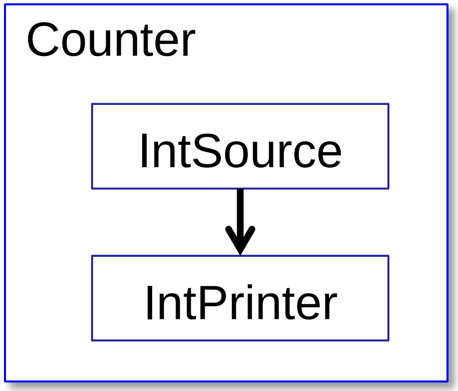
IntSource

IntPrinter

- A pipeline is declared by the keyword ‘**pipeline**’.
  - Represents a producer-consumer type of chain
- Keyword ‘**add**’ allows us to add filters to a pipeline.
  - filters appear in the order they are added to the pipeline

StreamIt example: a simple counter (cont.)

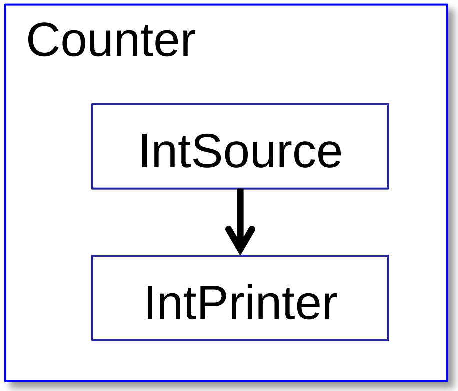
```
void→void pipeline Counter {
   add IntSource ();
   add IntPrinter ();
}

void→int filter IntSource () {
   int ctr;
   init { ctr = 0; }
   work push 1 {
      push (ctr++);
   }


int→void filter IntPrinter () {
   work pop 1 { print(pop()); }
}
```



- All StreamIt constructs are **typed**.
  - filters, pipelines, ...
- Input type
  - the type of data read from the input channel
- Output type
  - the type of data written to the output channel

StreamIt example: a simple counter (cont.)

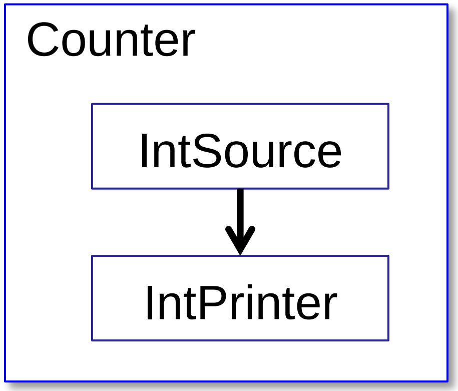
```
void→void pipeline Counter {
   add IntSource ();
   add IntPrinter ();
}

void→int filter IntSource () {
   int ctr;
   init { ctr = 0; }
   work push 1 {
      push (ctr++);
   }

int→void filter IntPrinter () {
   work pop 1 { print(pop()); }
}
```



- All StreamIt constructs are **typed**.
- Example:
  - void -> int
    - Nothing read from input (=src filter)
    - int-data written to output
  - void -> void
    - nothing read from input channel
    - nothing written on output channel
    - this is the program’s top-level construct.
      - similar to the `main()` function in C

StreamIt example: a simple counter (cont.)

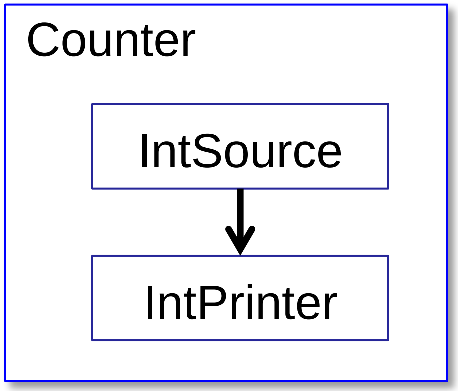
```
void→void pipeline Counter {
   add IntSource ();
   add IntPrinter ();
}

void→int filter IntSource () {
   int ctr;
   init { ctr = 0; }
   work push 1 {
      push (ctr++);
   }

int→void filter IntPrinter () {
   work pop 1 { print(pop()); }
}
```



- A filter's work() function can have local variables
  - Same as local variables in C.
- A filter can have an **init()** function.
  - Executed at program start, before filters start execution.
  - Mostly used to initialize variables.
  - Similar to a C++ constructor.

StreamIt example: a simple counter (cont.)

```
void→void pipeline Counter {
   add IntSource ();
   add IntPrinter ();
}

void→int filter IntSource () {
   int ctr;
   init { ctr = 0; }
   work push 1 {
      push (ctr++);
   }

int→void filter IntPrinter () {
   work pop 1 { print(pop()); }
}
```

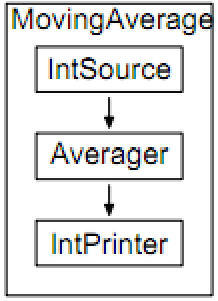


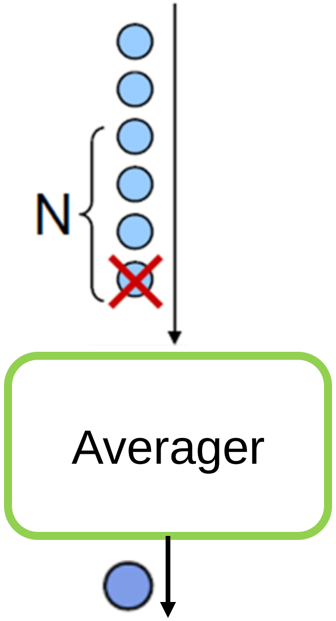
Compile and run:

```
$  strc –o Counter Counter.str
$  ./Counter –i  4
0
1
2
3
```

Example 2: Moving Average Filter

```
void→void pipeline MovingAverage {
  add IntSource ();
  add Averager (10);
  add IntPrinter ();
}

int→int filter Averager (int n) {
  work pop 1 push 1 peek n {
    int sum = 0;
    for ( int i = 0; i < n; i++)
      sum += peek(i);
    push (sum/n);
    pop();
  }
}
```



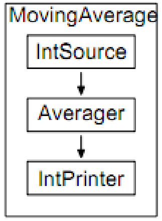


- Averager **peeks** N elements in for-loop and sums them up.
- Afterwards the moving average is **pushed** onto output stream.
- One element **popped** from input stream afterwards.

Example 2: Moving Average Filter

```
void→void pipeline MovingAverage {
    add IntSource ();
    add Averager (10);
    add IntPrinter ();
}

int→int filter Averager (int n) {
    work pop 1 push 1 peek n {
        int sum = 0;
        for ( int i = 0; i < n; i++)
            sum += peek(i);
        push (sum/n);
        pop();
    }
}
```

- Averager receives a **stream parameter n** that specifies the number of elements to average.
    - Value of **n** passed with the **add** statement that creates the filter.
- Within a filter, a stream parameter is a constant.
    - It is not possible to modify the constant, e.g., assigning n = 0.
    - The code in the work function may *read* the stream parameter.



Note: the StreamIt compiler ensures that the Averager only executes when at least n elements available on input.

Moving Average Filter in C

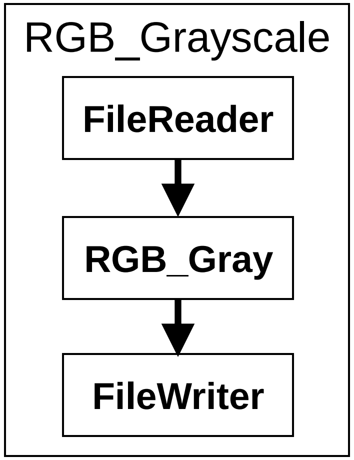
```
void Averager (
  int * src,
  int * dest,
  int * srcIndex,
  int * destIndex,
  int srcBufferSize,
  int destBufferSize,
  int n) {
                                                  //push (sum/n);
    int sum = 0;
    for ( int i = 0; i < n; i++)
      sum += src[(*srcIndex + i) % srcBufferSize]; //peek(i);
    dest[*destIndex] = sum/n;
    *destIndex = (*destIndex + 1) % destBufferSize;
    *srcIndex = (*srcIndex + 1) % srcBufferSize; //pop();
  }
}
```

- Implementing the Averager in C requires a lot of extra code:
  - input/output buffer management (in red)
  - scheduling of filters
  - synchronization
  - map onto hw
    - Cell SPEs?!
- The extra code clutters the C program!
- Programmer must commit to buffer implementation stratgegy, understand hw, ... ☹☹☹

Example 3: Grayscale Conversion

pipeline-parallelism!

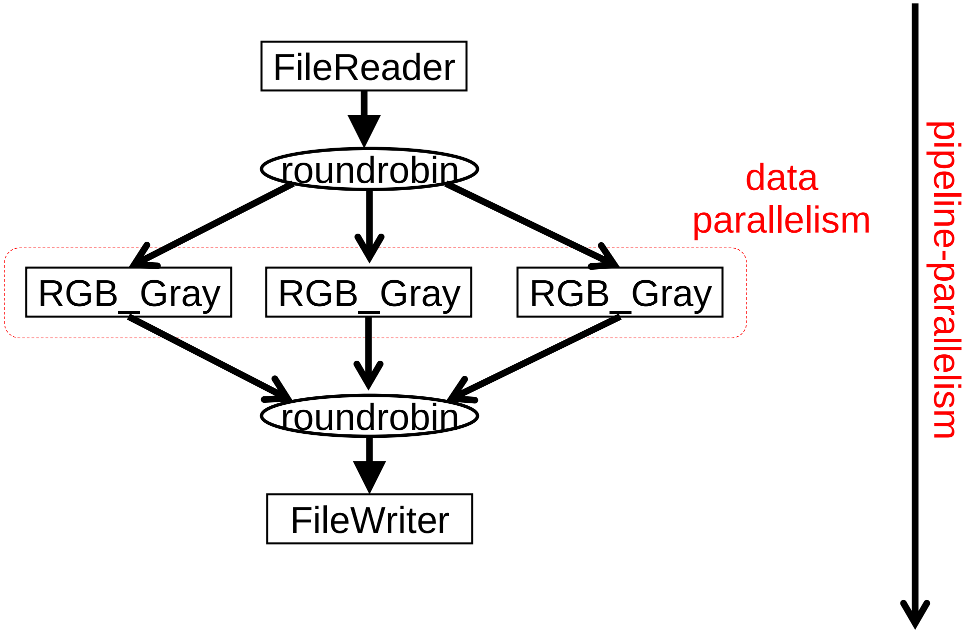
```
void -> void pipeline RGB_Grayscale {
   add FileReader< int >( "./in.ppm.bin" );
   add RGB_Gray;
   add FileWriter< int >(./out.ppm.bin" );
}

int -> int filter RGB_Gray {
   work push 1 pop 3 {
      int R = pop();
      int G = pop();
      int B = pop();
      int Gray = (int)(R*0.3 + G*0.59 + B*0.11);
      push(Gray);
   }
}
```



- FileReader is a source-filter that reads input (int in this case!) from given file and pushes items on output stream.
- FileWriter is a sink-filter that writes to a file.
- FileReader and FileWriter can be instantiated <...> for any StreamIt data type:
  - int, float, complex, bit

Example 3: Grayscale Conversion (cont.)

```
void -> void pipeline RGB_Grayscale {
   add FileReader< int >( "./in.ppm.bin" );
   add RGB_Gray;
   add FileWriter< int >(./out.ppm.bin" );
}

int -> int splitjoin Converter {
         split roundrobin (3);
         add RGB_Gray;
         join roundrobin;
}

int -> int filter RGB_Gray {
   work push 1 pop 3 {
      int R = pop();
      int G = pop();
      int B = pop();
      int Gray = (int)(R*0.3 + G*0.59 + B*0.11);
      push(Gray);
   }
}
```



- **Duplicate** RGB_Gray n times inside a splitjoin to introduce **data-parallelism**.
  - Compare this to our pthread-solution! ☺
- One limitation: because StreamIt is based on SDF (synchronous data-flow), the graph-structure must be fixed at compile-time!

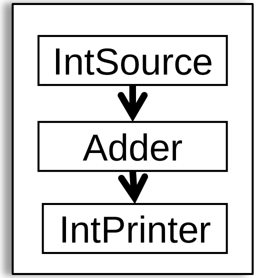
Data-Parallelism and Stateful Filters

```
void -> void pipeline GaussSeriesSum {
   add IntSource;
   add Adder;
   add IntPrinter;
}

int -> int filter Adder {
         int sum; // filter state information
         init { sum = 0; }
         work pop 1 push 1 {
             sum = sum + pop();
             push (sum);
         }
}

int -> void filter IntPrinter {
   work pop 1 { print (pop ()); }
}
```

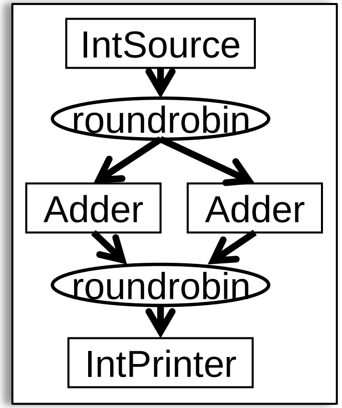
- Suppose we want to compute the sum of the arithmetic series 1, 2, 3, 4, 5, ...
- In C:

```
unsigned int sum = 0;
for (int i = 1; i < ...; i++) {
    sum += i; printf("%d, ");
}
```

- The Adder remembers the temporary sum between work-function invocations!
  - sum is state-information!
  - We cannot duplicate Adder without changing the program semantics:



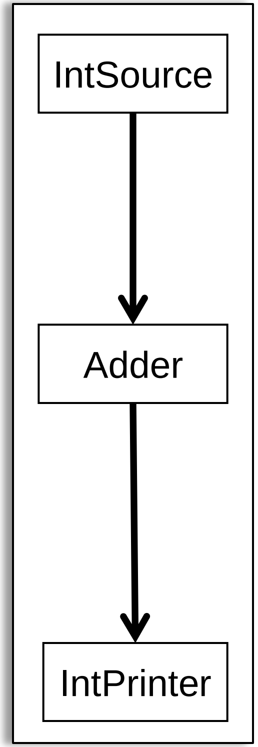
prints 1, 3, 6, ...



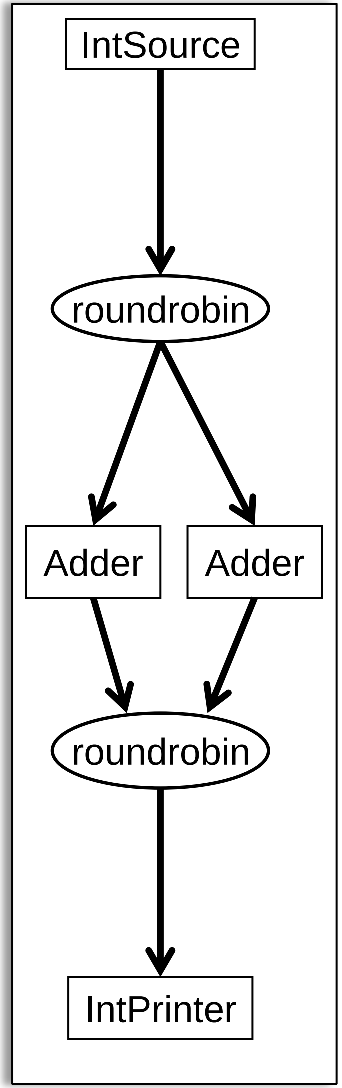
prints 1, 2, 4, ...

Data-Parallelism and Stateful Filters (cont.)

```
void -> void pipeline GaussSeriesSum {
   add IntSource;
   add Adder;
   add IntPrinter;
}

int -> int filter Adder {
         int sum; // filter state information
         init { sum = 0; }
         work pop 1 push 1 {
             sum = sum + pop();
             push (sum);
         }
}

int -> void filter IntPrinter {
   work pop 1 { print (pop ()); }
}
```



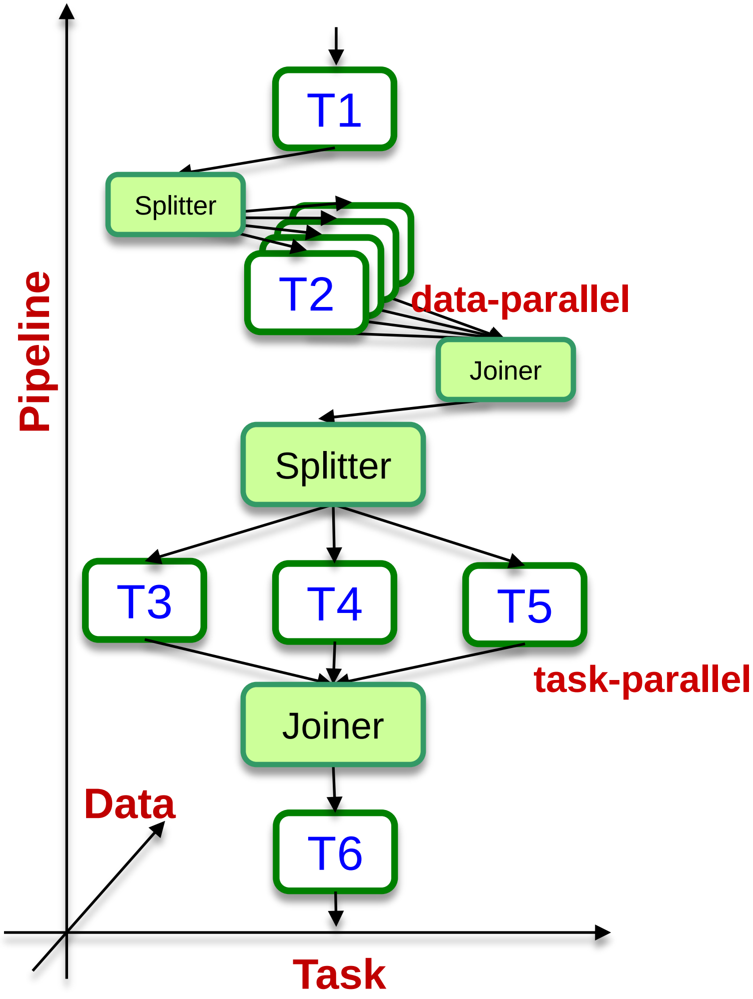
prints 1, 3, 6, ...



prints 1, 2, 4, ...

Therefore: only stateless filters can be duplicated!

StreamIt: Task+Data+Pipeline Parallelism



Data Parallelism

- Same operation on different data items
- Placed within splitter/joiner pair (duplication)
  - e.g., 4 x T2

Task Parallelism

- Between filters *without* producer/consumer relationship
  - e.g., T3, T4, T5

Pipeline Parallelism

- Between producer-consumer pairs
  - e.g., T1, T2, …