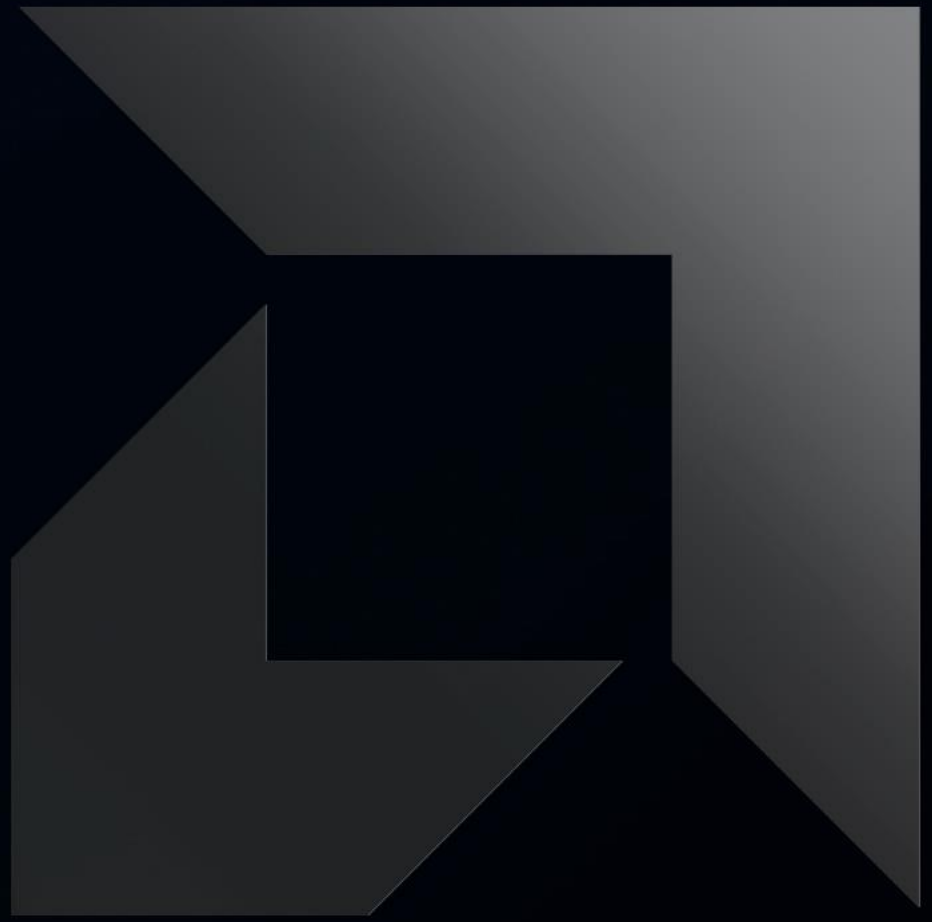
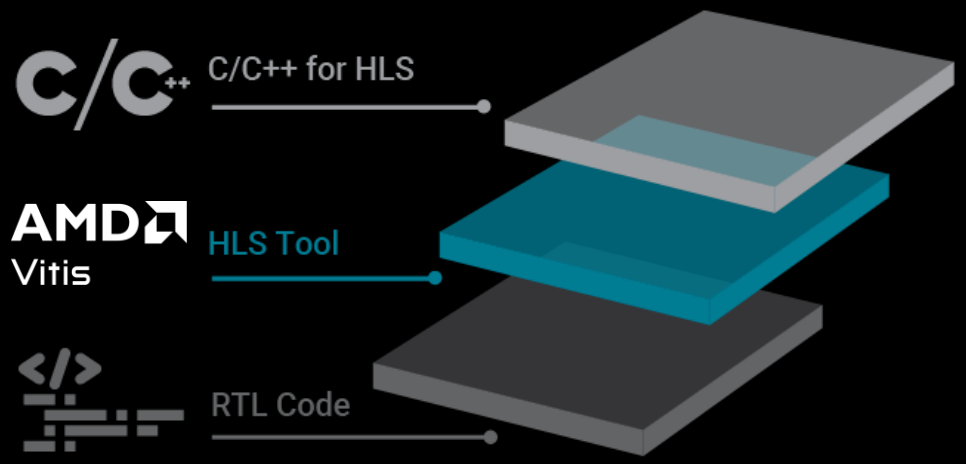


High-Performance AMD Vitis™ HLS Design with Task-Level Parallelism

AMD
together we advance_

Introduction to Vitis High-Level Synthesis

Automated design process that converts high-level abstraction languages (such as C/C++ to RTL)



- Faster design iteration and faster design verification
- Easy to retarget to different hardware with minimal changes to source code
- Generated RTL can be used as an IP directly within the Vivado™ tool or Vitis™ Model Composer
- Generated kernels can be imported into Vitis IDE
- 600+ library functions that are ready to use
- Support for parallel programming constructs to model desired implementation

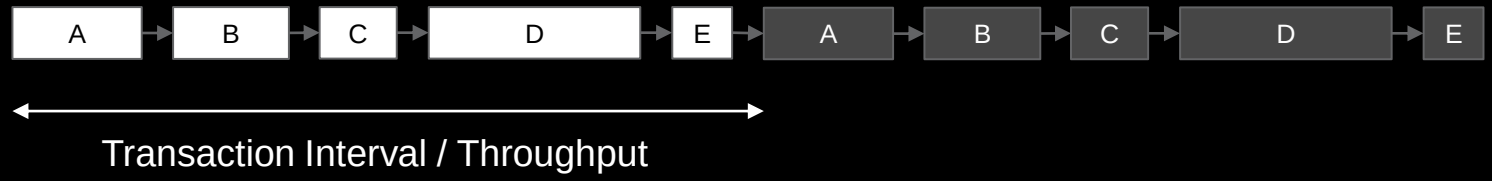
Agenda

1. Task-Level Parallelism Overview
2. Tasks
3. Channels

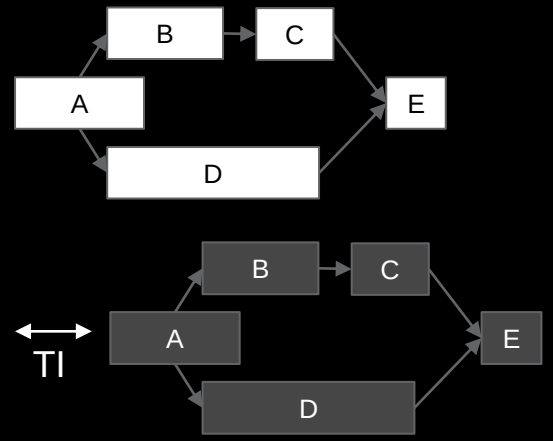
Task-Level Parallelism

Why Task-Level Parallelism (TLP)?

C/C++ for CPU



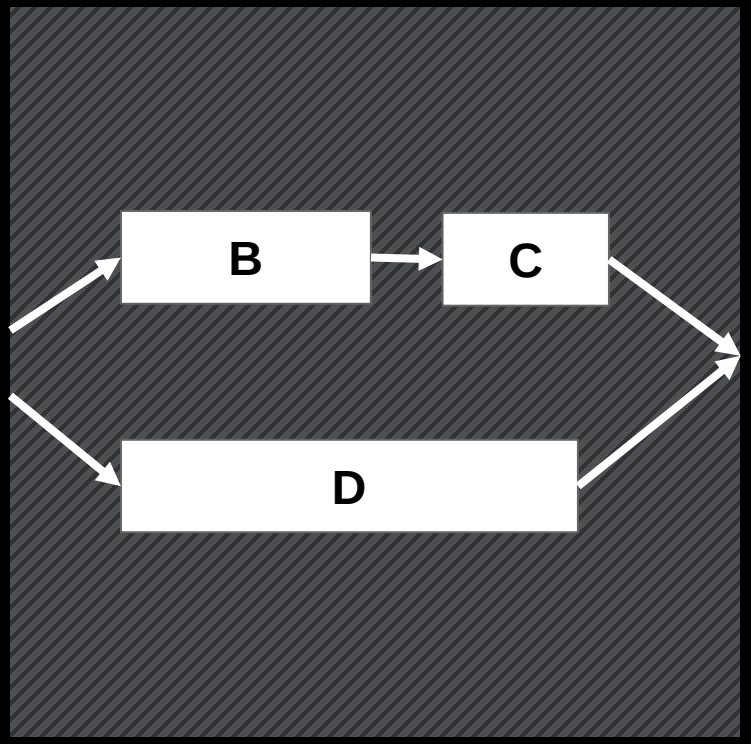
C/C++ for FPGA



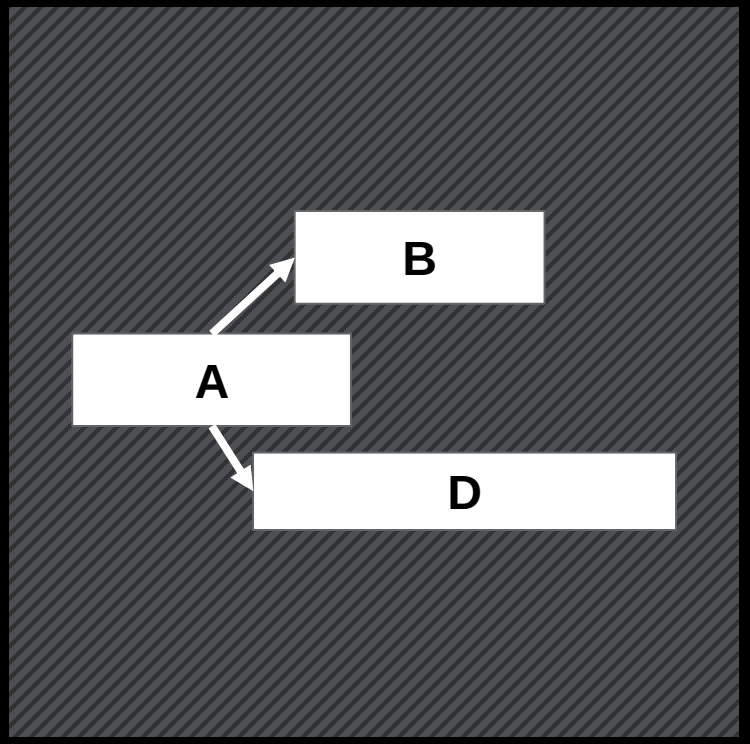
C/C++ code targeting Vitis™ HLS **must** be coded with a task-parallel architecture

- Performance
- Efficiency
- Synthesizability
- Timing closure

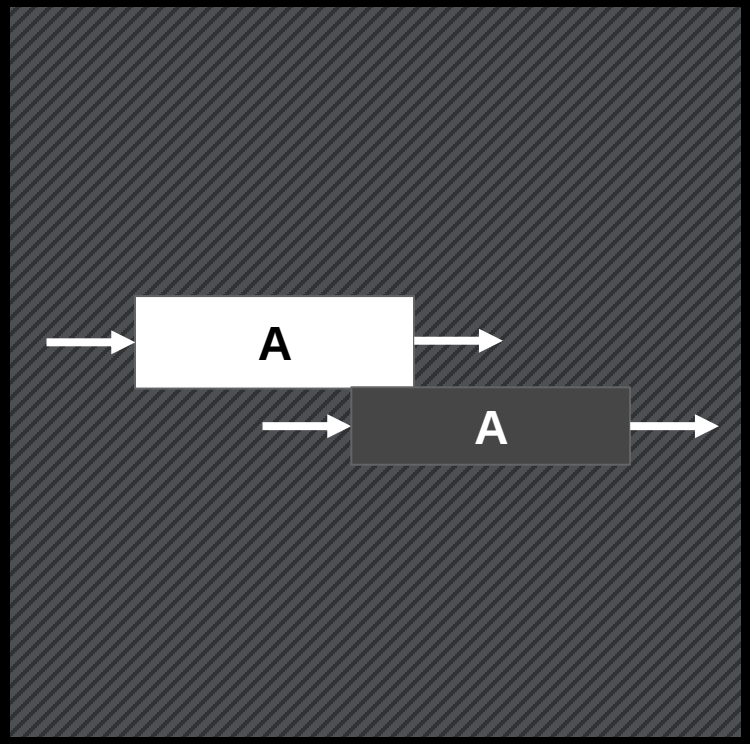
Different Kinds of Parallelism



Two tasks with independent datapaths can execute in parallel

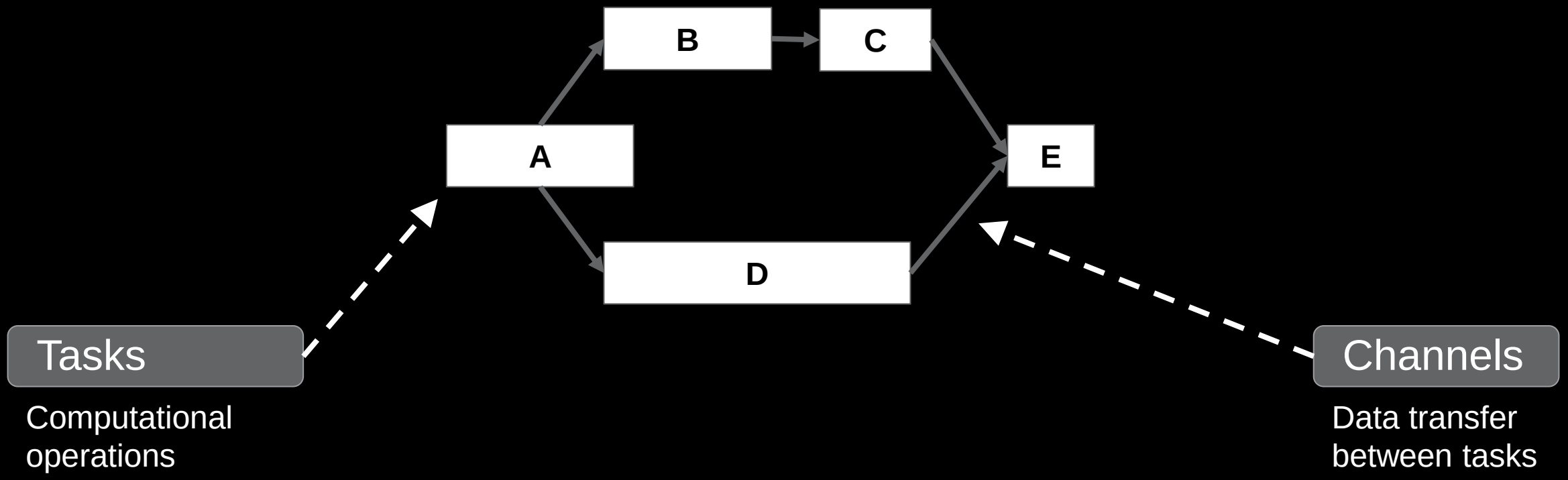


Subsequent tasks begin execution before previous tasks complete



One task can begin processing next iteration before current iteration has completed

Tasks and Channels



Tasks

Two Types of Task-Level Parallelism

Control Driven

- Tool automatically generates task-level parallel region from sequential functions when the dataflow pragma is added
- Block-level control signals are generated that explicitly start and stop each process

Data Driven

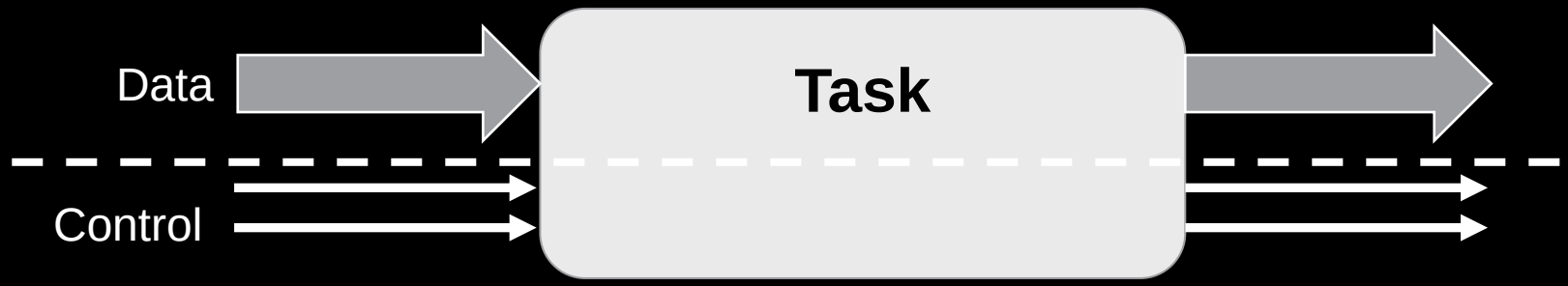
- User explicitly dictates task-level parallelism
- Each task executes only in response to streaming data availability, like an infinite loop

Example of Control-Driven Task-Level Parallelism

```
void diamond(data_t vecIn[N], data_t vecOut[N]) {
  data_t c1[N], c2[N], c3[N], c4[N];
#pragma HLS dataflow
  funcA(vecIn, c1, c2);
  funcB(c1, c3);
  funcC(c2, c4);
  funcD(c3, c4, vecOut);
}

void funcA(data_t *in, data_t *out1, data_t *out2) {
 for (int i = 0; i < N; i++) {
    data_t t = in[i] * 3;
    out1[i] = t;
    out2[i] = t;
  }
}
…
```

Definition of a Control-Driven Task



- Control preserves sequential semantics and enables task synchronization
- Control is required for external memory interface access

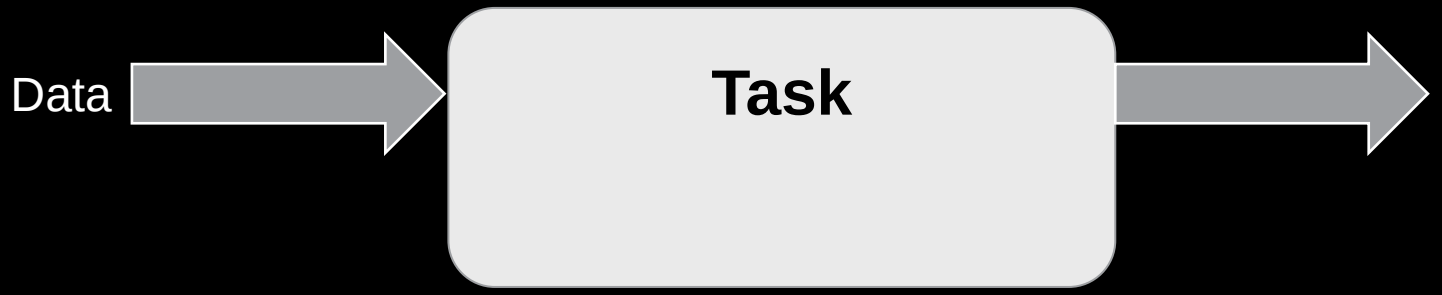
Example of Data-Driven Task-Level Parallelism

```
void diamond( hls::stream<data_t> &vecIn,
              hls::stream<data_t> &vecOut) {
hls::stream<data_t> c1,c2,c3,c4;

  hls_thread_local hls::task taskA(funcA,vecIn, c1, c2);
  hls_thread_local hls::task taskB(funcB,c1,c3);
  hls_thread_local hls::task taskC(funcC,c2,c4);
  hls_thread_local hls::task taskD(funcD,c3,c4,vecOut);
}

void funcA( hls::stream<data_t> &in,
            hls::stream<data_t> &out1
            hls::stream<data_t> &out2) {
    data_t t = in.read();
    out1.write(t);
    out2.write(t);
}
…
```

Definition of a Data-Driven Task



- Task execution is entirely driven by the data, which must be purely streaming
- Tasks run infinitely-simply reading from the input when data is available, processing the data, and writing to output when space is available

Benefits of Control-Driven and Data-Driven TLP

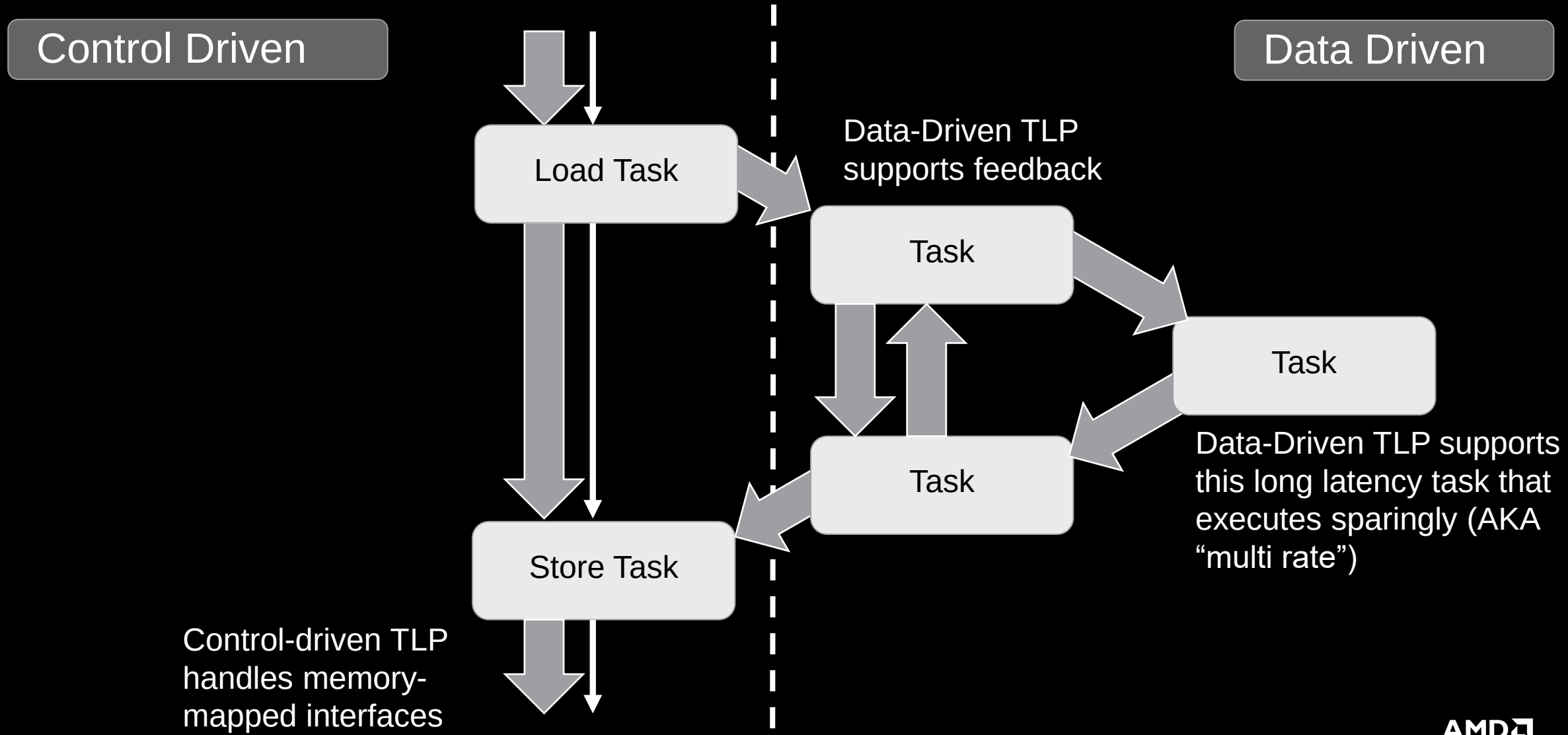
Control Driven

- Tool automatically infers task-level parallelism
- Allows memory-mapped data transfers in TLP regions

Data Driven

- More intuitive design style for purely streaming designs
- Enables a new class of designs
  - Designs requiring feedback
  - Data-dependent multi-rate behavior
- Provides additional C simulation capabilities
  - Concurrency simulation
  - Deadlock detection

Combining Control-Driven and Data-Driven TLP



Channels

Overview of Channels

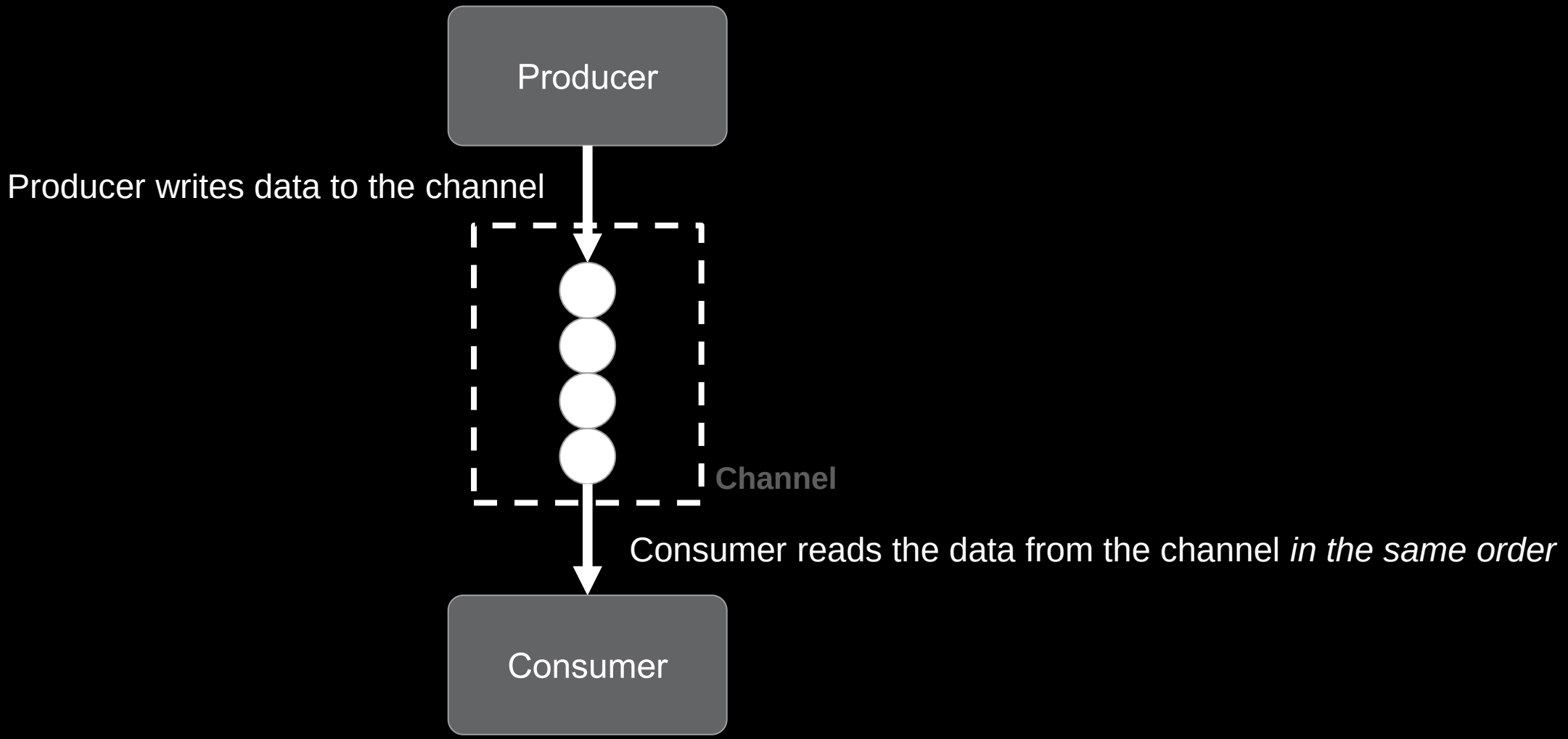
HLS::Stream

- Sequential data transfer channel
  - Producer is allowed write access
  - Consumer is allowed read access

HLS::Stream_of_Blocks

- Supports data transfers with complex memory access patterns
- User has explicit control of control of blocks within channel and sharing between tasks

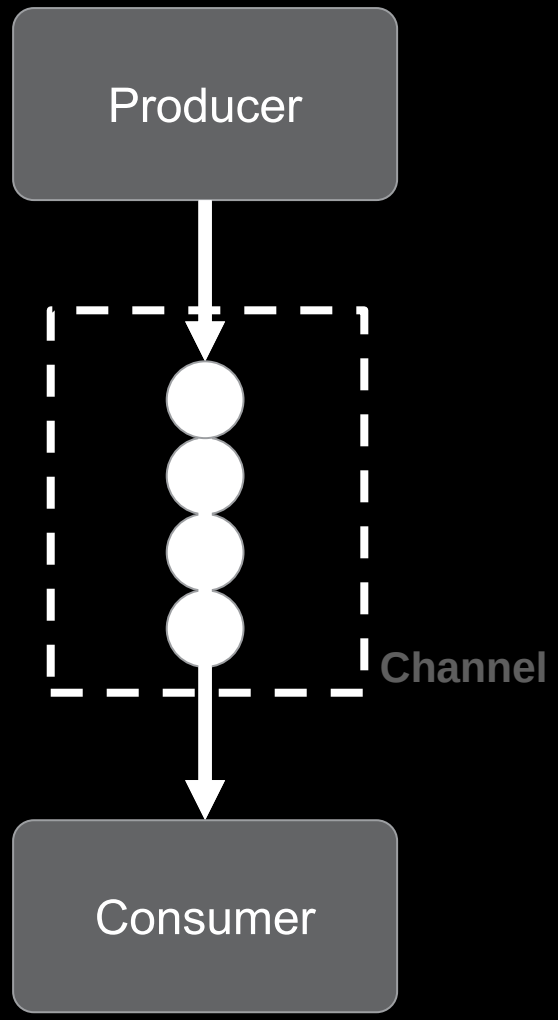
First in First Out (FIFO)



Overview of HLS::Stream

- HLS::Stream is an HLS implementation of a FIFO
- Provides channel write and read accessor functions to the producer and consumer tasks, respectively
  - Also provides additional APIs:
  - `.empty(), .full(), .size(), and .capacity()`

```
void funcA( hls::stream<data_t> &in,
            hls::stream<data_t> &out1
            hls::stream<data_t> &out2) {
    data_t t = in.read();
    out1.write(t);
    out2.write(t);
}
```

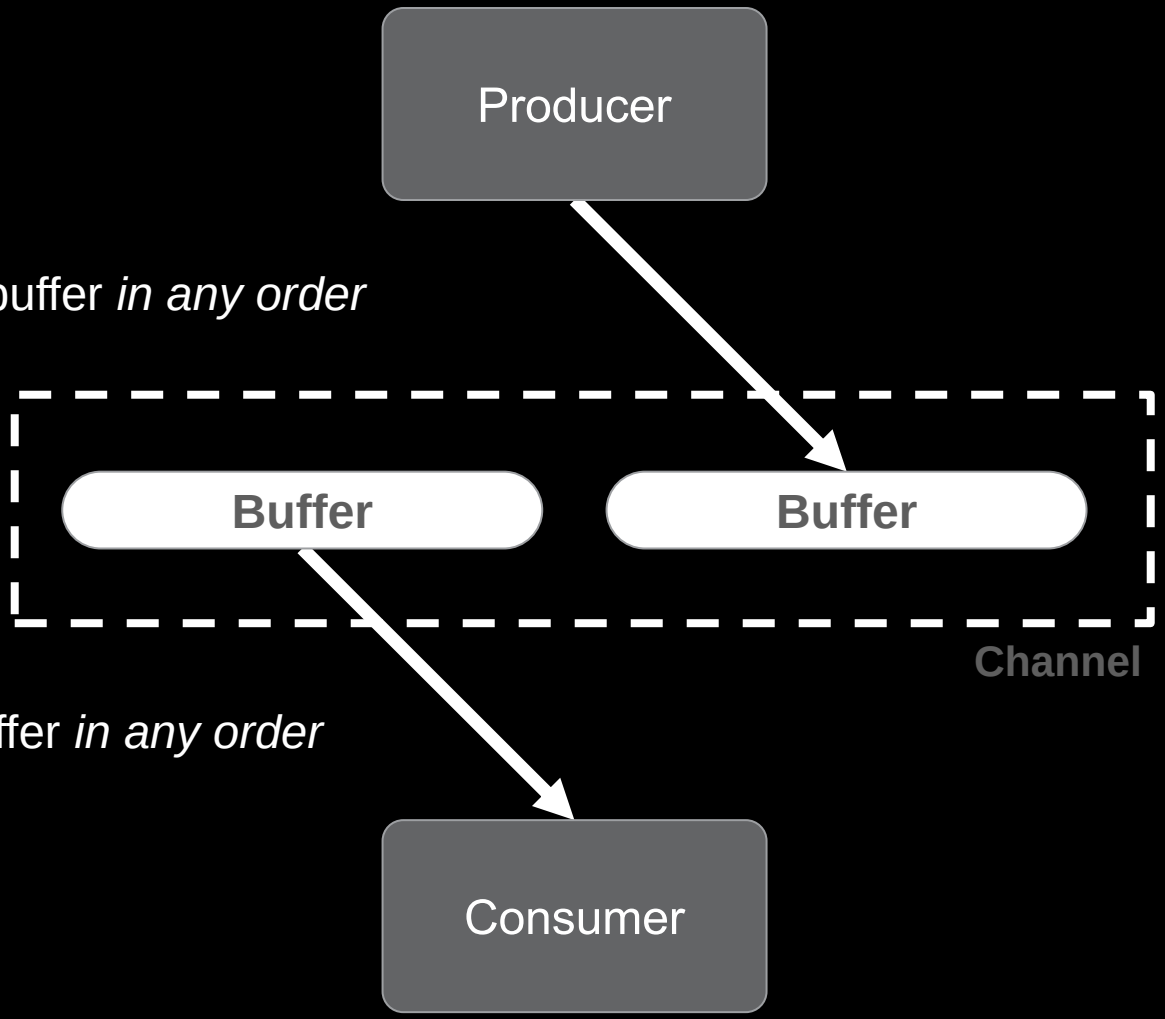


Parallel In Parallel Out (PIPO)

Producer writes data to the channel buffer *in any order*

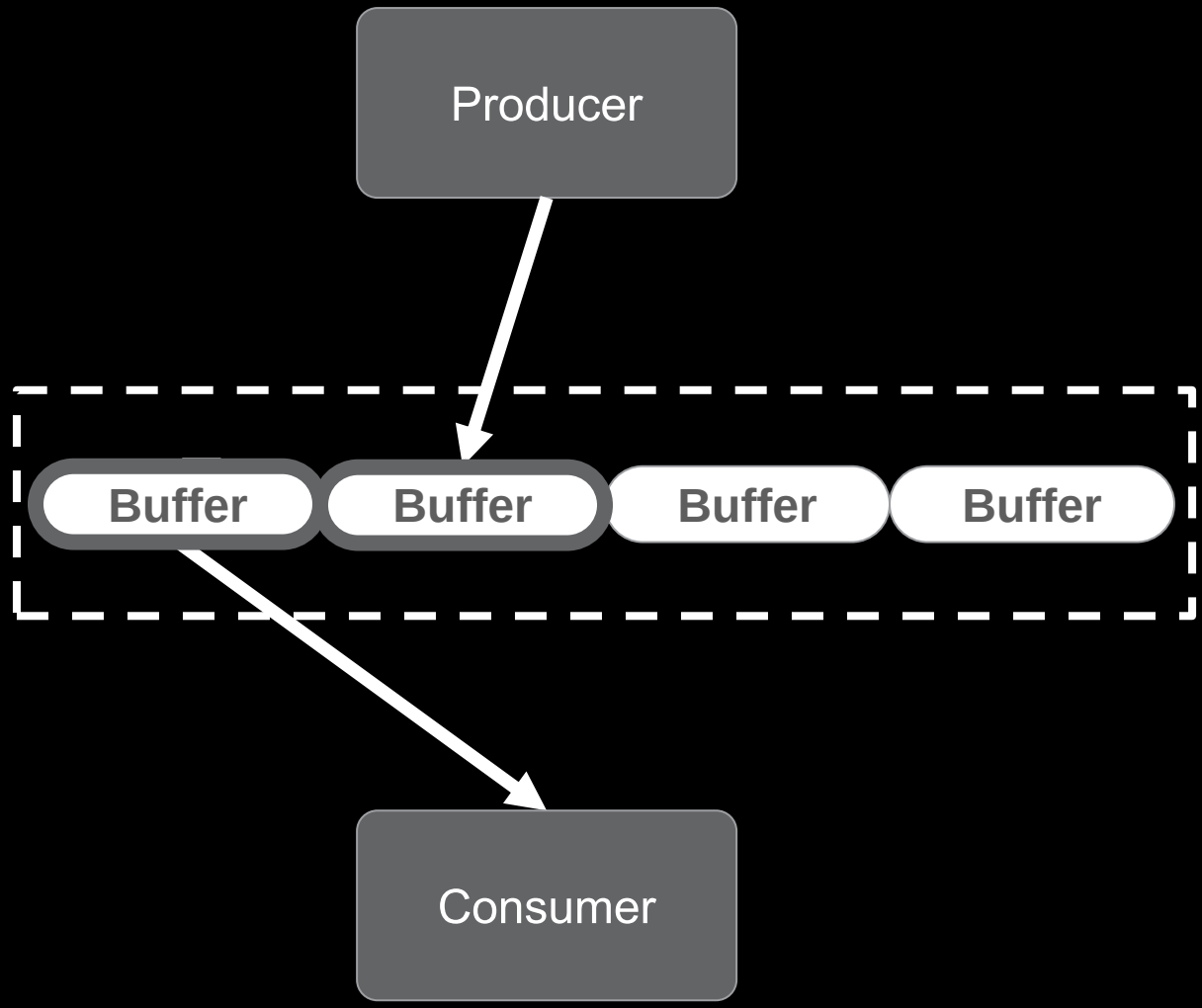
Data is passed from producer to consumer by swapping control of the two buffers

Consumer reads the data from the channel buffer *in any order*



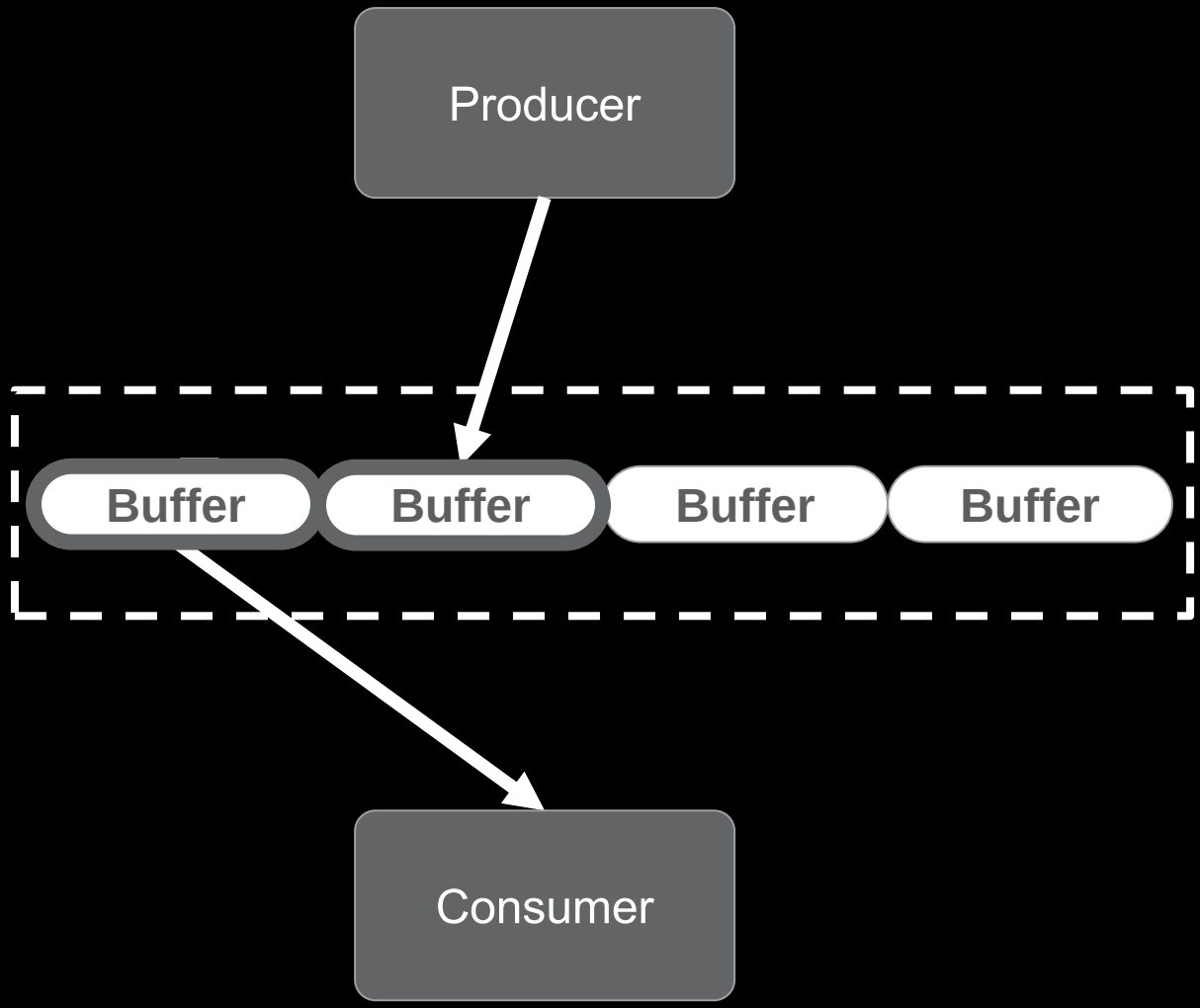
Overview of HLS::Stream_of_Blocks

- Explicit channel management
  - Mutex-like locks are used to acquire the next block from the stream for data access
- Allows for random access of each block while streaming a set of blocks



Example of HLS::Stream_of_Blocks

```
void funcB( hls::stream_of_blocks<block_data_t> &in,
            hls::stream_of_blocks<block_data_t> &out) {
for (int i = 0; i < N/NUM_BLOCKS; i++) {
    hls::read_lock<block_data_t> inL(in);
    hls::write_lock<block_data_t> outL(out);
    for (unsigned int j = 0; j < NUM_BLOCKS; j++)
       outL[inL[j]] = j + 25;
    }
}
```



Channels Summary

HLS::Stream

- Implements a FIFO
- Sequential data access
- Abstractions for read and write to stream

HLS::Stream_of_Blocks

- User-controlled sharing of blocks between tasks
- Random access is allowed within a block

Demo

Vitis HLS 2022.2 - dataflow (C:\Users\nmoeller\Documents\Presentation\dataflow)

File Edit Project Solution Window Help

Explorer | Module Hierarchy

- dataflow
  - Includes
  - Source
    - diamond.cpp
    - diamond.h
  - Test Bench
  - NO_TLP
    - constraints
    - impl
    - syn
  - ORIGINAL
  - REFACTOR
  - STREAM

Flow Navigator

- C SIMULATION
  - Run C Simulation
  - Reports & Viewers
- C SYNTHESIS
  - Run C Synthesis
  - Reports & Viewers
    - Report
    - Function Call Graph
    - Schedule Viewer
    - Dataflow Viewer
- C/RTL COSIMULATION
  - Run Cosimulation
  - Reports & Viewers
- IMPLEMENTATION
  - Export RTL
  - Run Implementation
  - Reports & Viewers

diamond.cpp | diamond_csim.log | diamond.h | Synthesis Summary(STREAM) | Synthesis Summary(NO_TLP)

```
24
25  #ifdef ORIGINAL
26
27  void diamond(data_t vecIn[N], data_t vecOut[N])
28  {
29    data_t c1[N], c2[N], c3[N], c4[N];
30  #pragma HLS dataflow
31    funcA(vecIn, c1, c2);
32    funcB(c1, c3);
33    funcC(c2, c4);
34    funcD(c3, c4, vecOut);
35  }
36
37  void funcA(data_t *in, data_t *out1, data_t *out2)
38  {
39  #pragma HLS inline off
40  Loop0:
41    for (int i = 0; i < N; i++)
42    {
43  #pragma HLS pipeline
44      data_t t = in[i] * 3;
45      out1[i] = t;
46      out2[i] = t;
47    }
48  }
49
50  void funcB(data_t *in, data_t *out)
51  {
52  #pragma HLS inline off
53  Loop0:
```

Outline | Directive

- diamond.h
- diamond(data_t[], data_t[]) : void
- funcA(data_t*, data_t*, data_t*) : void
- funcB(data_t*, data_t*) : void
- funcC(data_t*, data_t*) : void
- funcD(data_t*, data_t*, data_t*) : void
- funcA(data_t*, hls::stream<data_t>&, h
- funcB(hls::stream<data_t>&, hls::strean
- funcC(hls::stream<data_t>&, hls::strean
- funcD(hls::stream<data_t>&, hls::strear
- diamond(data_t[], data_t[]) : void
- funcA(hls::stream<data_t>&, hls::strean
- funcB(hls::stream<data_t>&, hls::strean
- funcC(hls::stream<data_t>&, hls::strean
- funcD(hls::stream<data_t>&, hls::strear
- diamond(hls::stream<data_t>&, hls::str

Console | Errors | Warnings | Guidance | Properties | Man Pages | Git Repositories | Modules/Loops

32 Guidance-Infos | 0 Guidance-War... | 0 Guidance-E...

Name	Web Help	Details
All Categories		
SCHEDULE		
[HLS 200-1470]		Pipelining result : Target II = NA, Final II = 1, Depth = 2, loop 'Loop0'
[HLS 200-1470]		Pipelining result : Target II = NA, Final II = 1, Depth = 2, loop 'Loop0'

STREAM | NO_TLP

Writable | Smart Insert | 42 : 4 [149]

1:27 PM 4/28/2023

Task Level Parallelism Summary

- Allow users to properly express parallelism
- Key to achieving performance on real world HLS Designs
- Extends HLS capabilities to address a larger class of designs
- Implement a larger portion of your design in one HLS IP block

Maximize the Benefits of HLS

Recommended Next Steps

- **New** Vitis HLS landing page

https://www.xilinx.com/products/design-tools/vitis/vitis-hls.html

  - Vitis HLS user guide
  - Vitis HLS training
  - Vits HLS forums
  - GitHub Vits HLS tool introductory examples

Questions


AMD

Demo