Memory and Data Movement Optimization

Kang Zhao

zhaokang@bupt.edu.cn

Outline

- Burst mode
- Wide bus transaction
- Double Buffer
- Streaming

Burst Mode for Continuous Memory Access

32 bit

```
void test( FIX_TYPE A[100][100], ...) {
#pragma HLS interface m_axi port=A offset=slave bundle=mem

    FIX_TYPE A_local[100][100];

    for(int i = 0; i < 100; i++) {
        for(int j = 0; j < 100; j++) {
            A_local[i][j] = A[i][j];
        }
    }

    ...
}
```

Multiple burst reads of length 10000 and bit width 32 in loop xxx has been inferred on port 'mem'

One data, one cycle

Bad Practice

```
for(int i = 0; i < 100; i++) {
    for(int j = 0; j < 100; j+=2) {
        A_local[i][j] = A[i][j];
    }
}
```

> 12000 cycles
(expected 5000)

Do not burst because memory is not continuous

```
for(int j = 0; j < 100; j++) {
    for(int i = 0; i < 100; i++) {
        A_local[i][j] = A[i][j];
    }
}
```

> 20000 cycles
(expected 10000)

Wider Bus Transaction

32 bit

```
void test( FIX_TYPE A[100][100], ...) {
#pragma HLS interface m_axi port=A offset=slave bundle=mem

    FIX_TYPE A_local[100][100];

    for(int i = 0; i < 100; i++) {
        for(int j = 0; j < 100; j++) {
            A_local[i][j] = A[i][j];
        }
    }

    ...
}
```

Multiple burst reads of length 10000 and bit width 32 in loop xxx has been inferred on port 'mem'

One 32-bit data per cycle

AXI bus is 512 bit wide – 15/16 bandwidth is wasted!

Wider Bus Transaction

```
typedef ap_uint<320> MEM_TYPE;

void test( MEM_TYPE A[100*10], ...) {
#pragma HLS interface m_axi port=A offset=slave bundle=mem

    FIX_TYPE A_local[100*100];
#pragma HLS array_partition variable=A_local cyclic factor=10

    for(int i = 0; i < 100*10; i+=10) {
#pragma HLS pipeline
       MEM_TYPE data = A[i];
       for(int ii = 0; ii < 10; ii++) {
          A_local[i*10 + ii] = data.range(0 + (ii*32), 31 + (ii*32));
       }
    }}
   ...
}
```

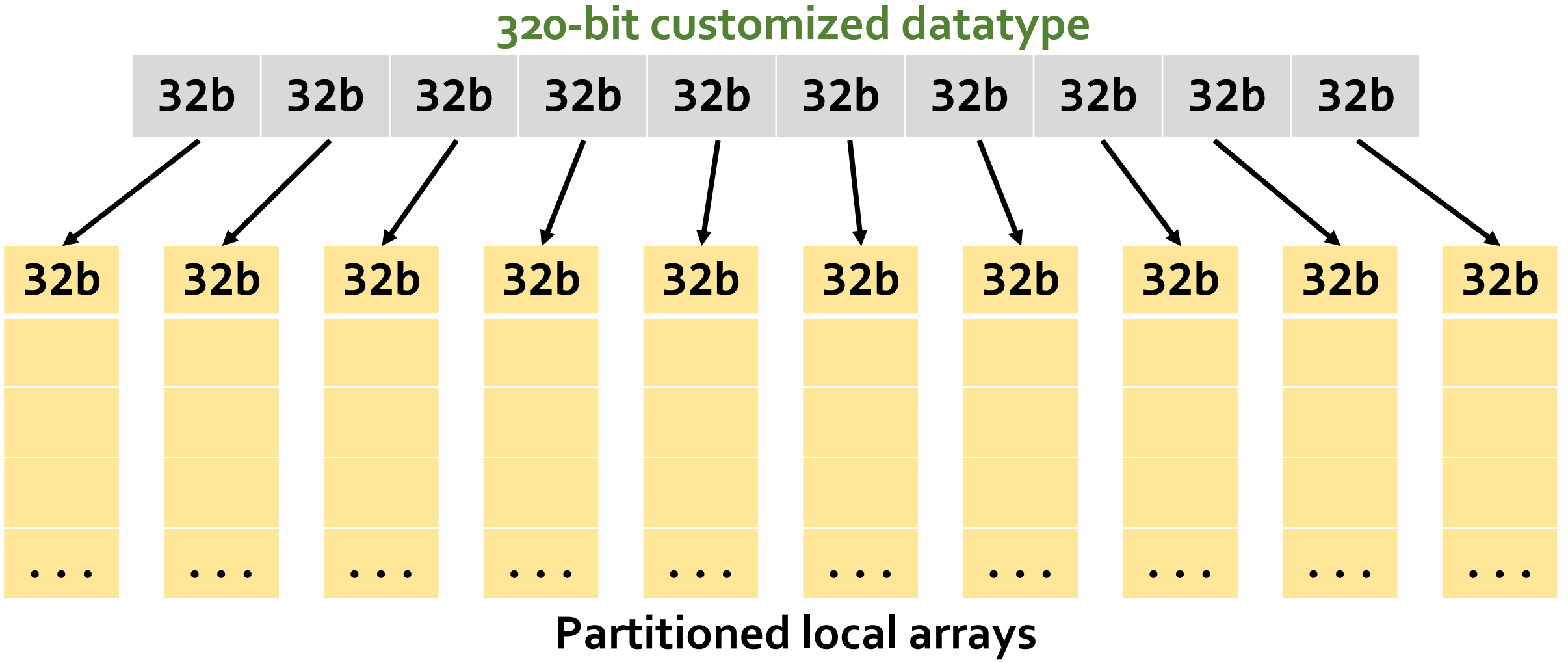
Create a customized wide data type – 320 bit

Reorganize into 1D array

Read 320 bit at each cycle

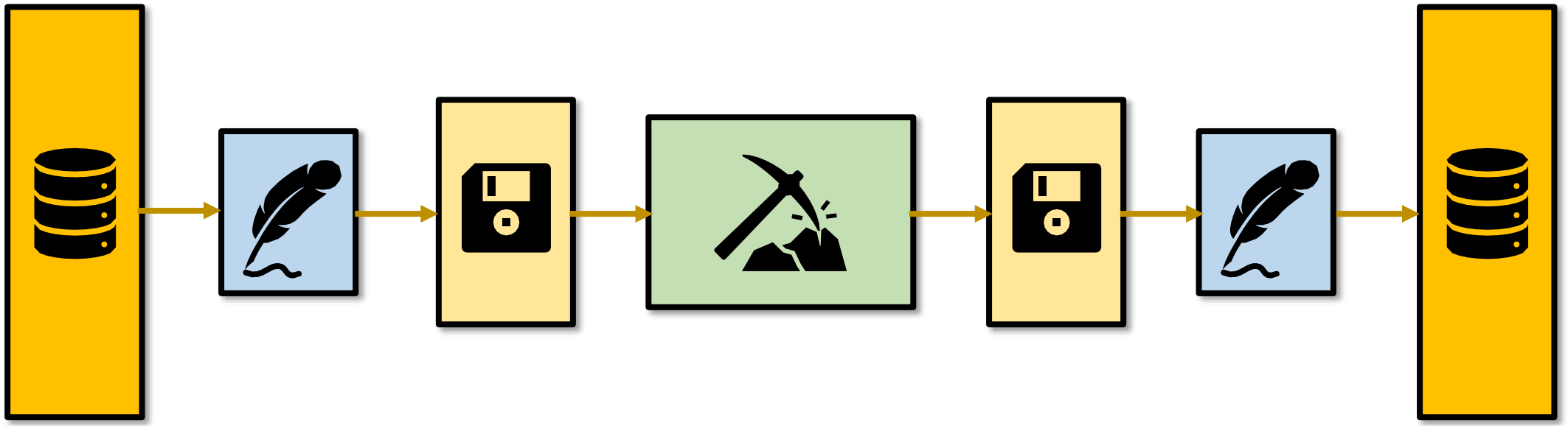
[INFO] Multiple burst reads of length 1000 and bit width 512

Wider Bus Transaction



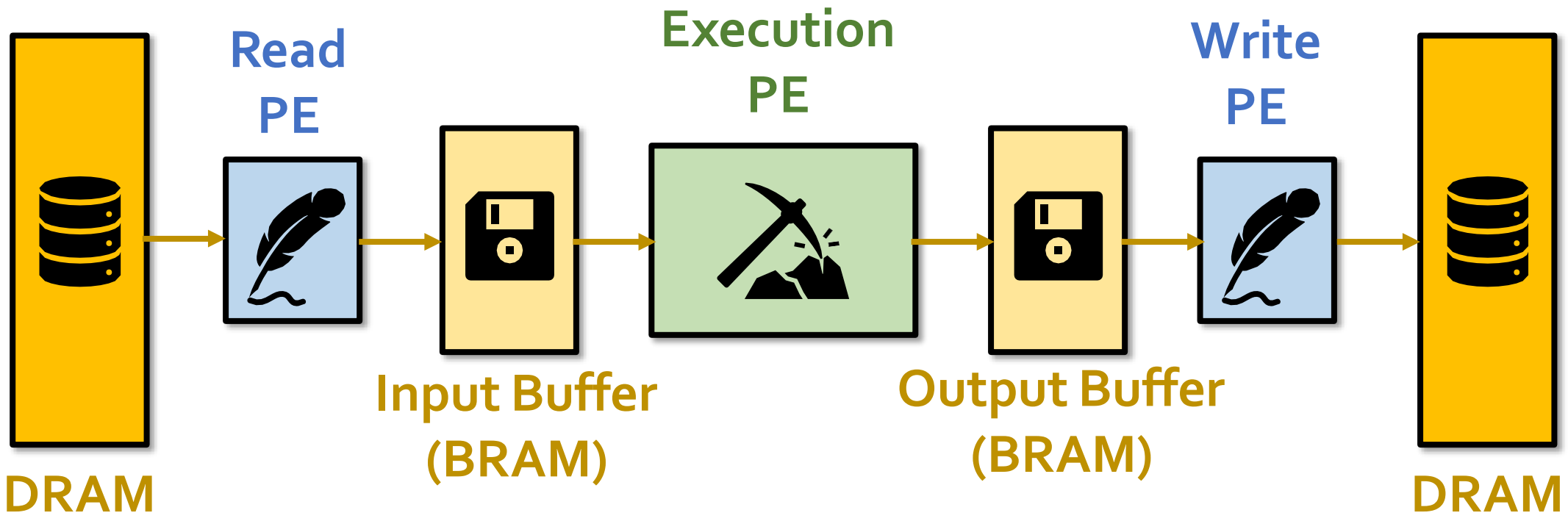
Single Buffer Limitation

- **Read-Execute-Write create dependency**



Single Buffer Limitation

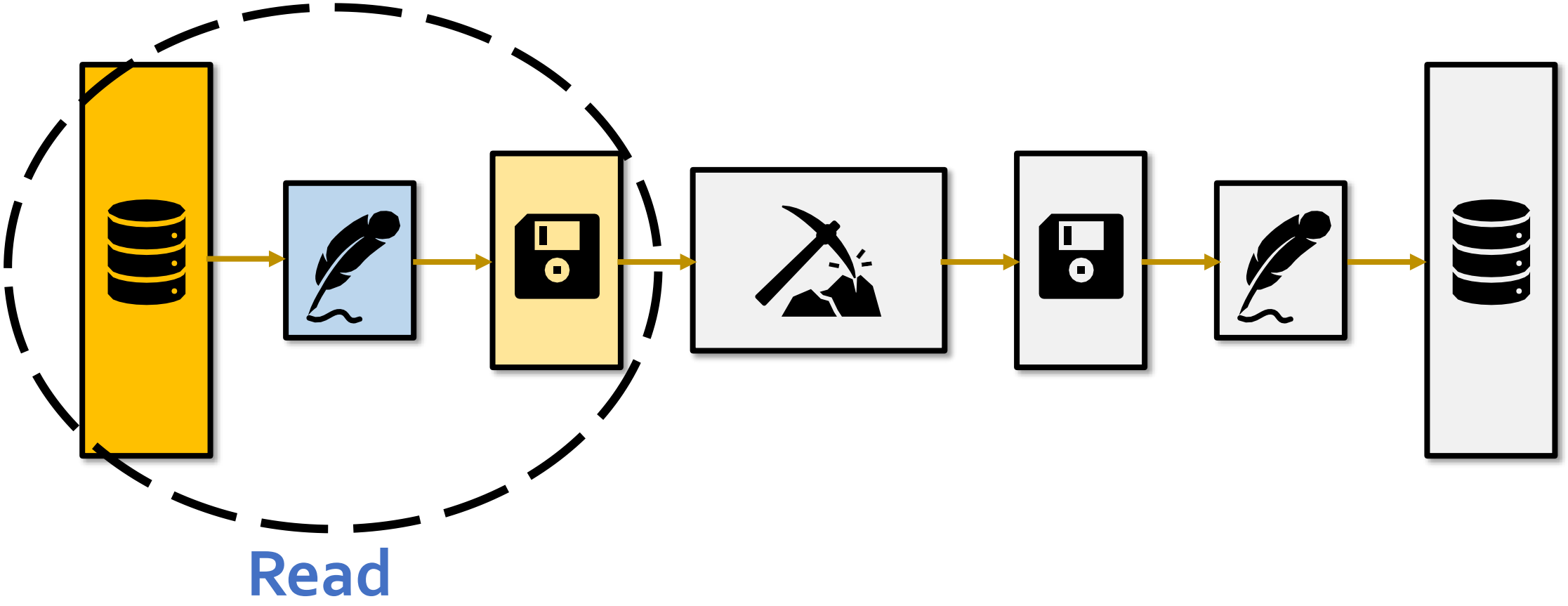
- **Read-Execute-Write create dependency**



**PE: processing element*

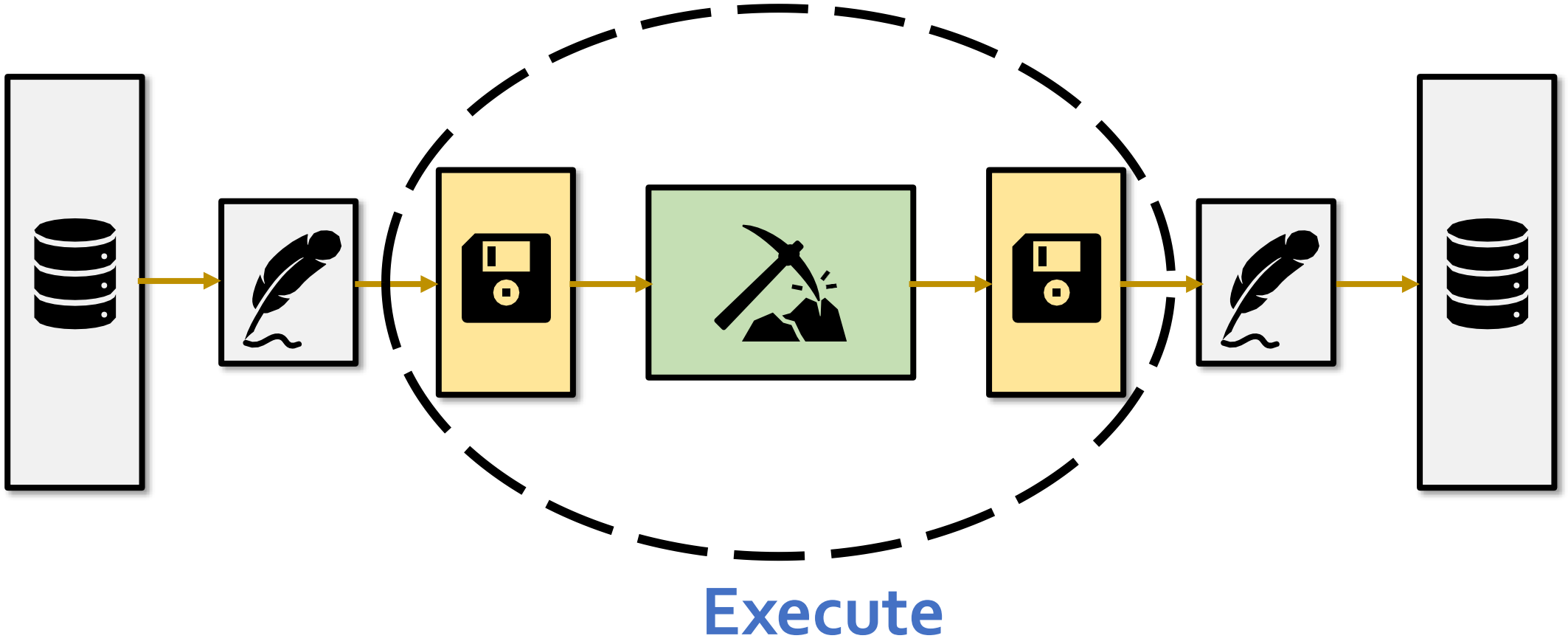
Single Buffer Limitation

- Read-Execute-Write create dependency



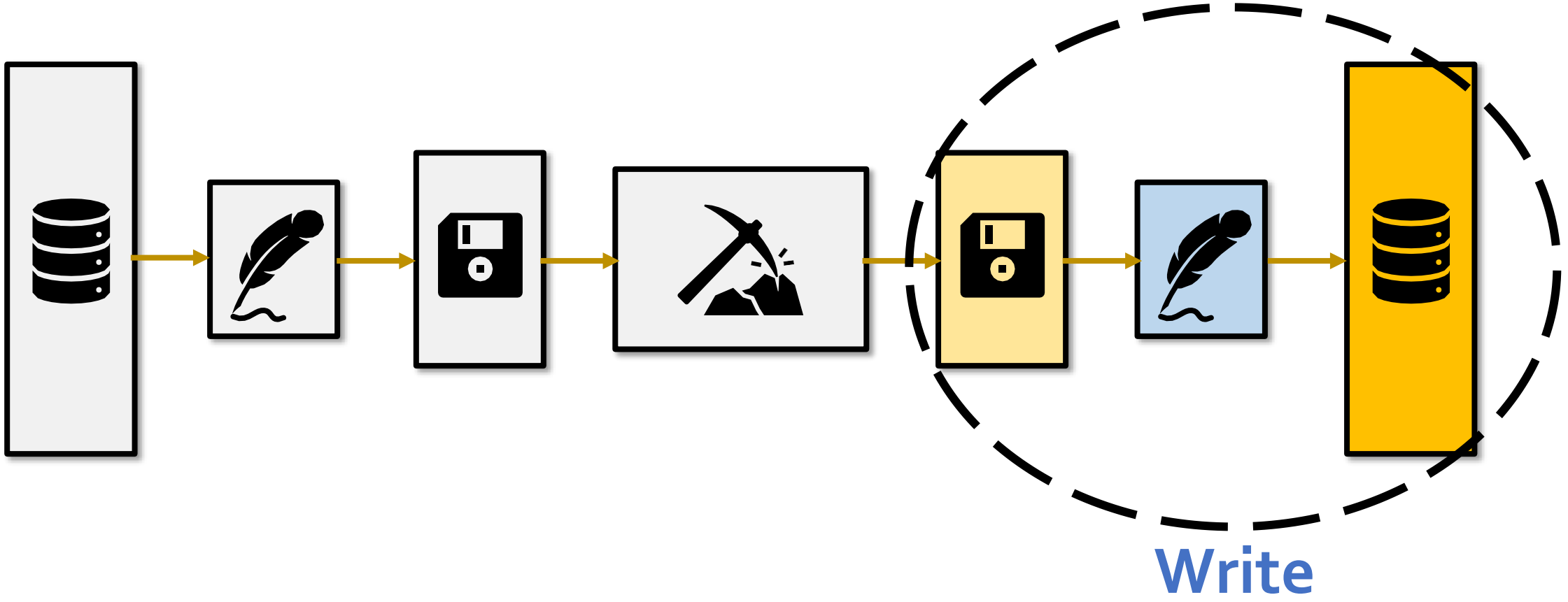
Single Buffer Limitation

- **Read-Execute-Write create dependency**



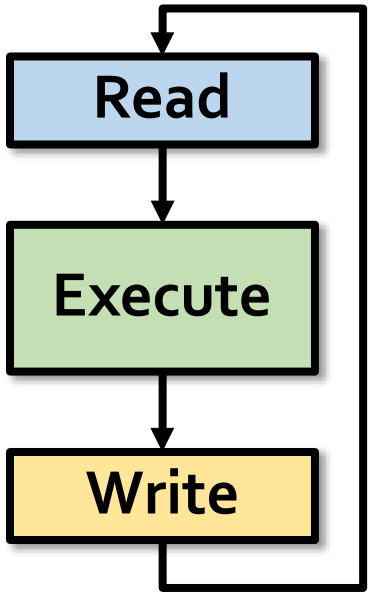
Single Buffer Limitation

- **Read-Execute-Write create dependency**

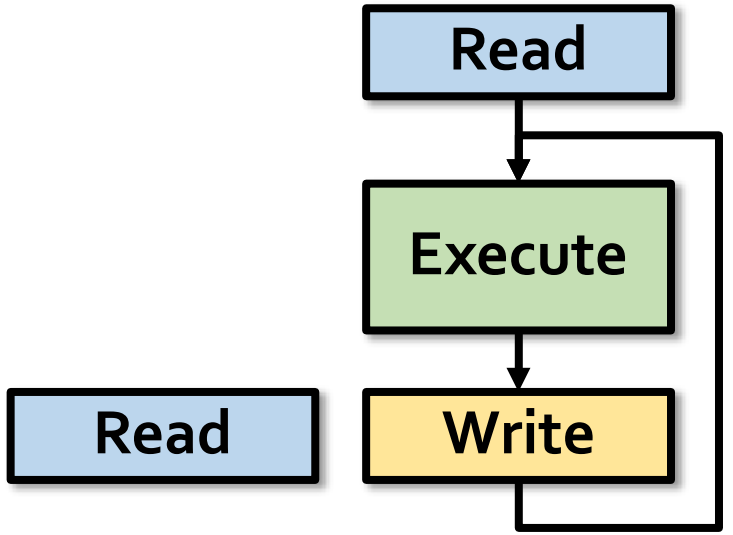


Overlap Read and Write (Pre-fetching)

```
for(int i = 0; i < N; i++) {
   read(buf_A, i);
   execute(buf_A, buf_B);
   write(buf_B, i);
}
```

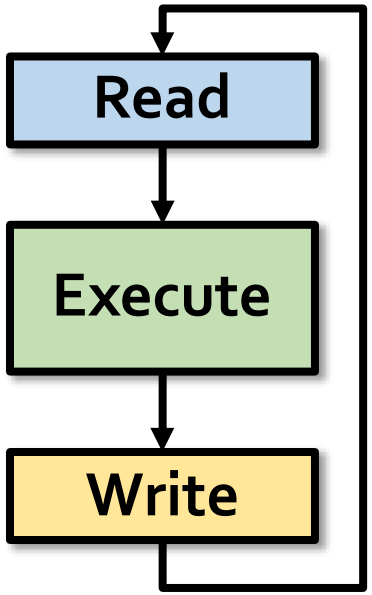



```
read(buf_A, i);
for(int i = 0; i < N; i++) {
   execute(buf_A, buf_B);
   write(buf_B, i);
   if(i < N-1) read(buf_A, i+1);
}
```



Overlap Read and Write (Pre-fetching)

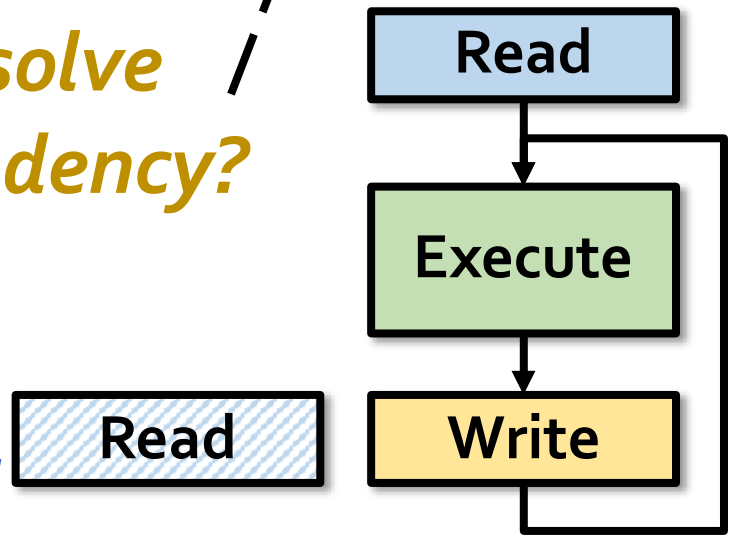
```
for(int i = 0; i < N; i++) {
    read(buf_A, i);
    execute(buf_A, buf_B);
    write(buf_B, i);
}
```



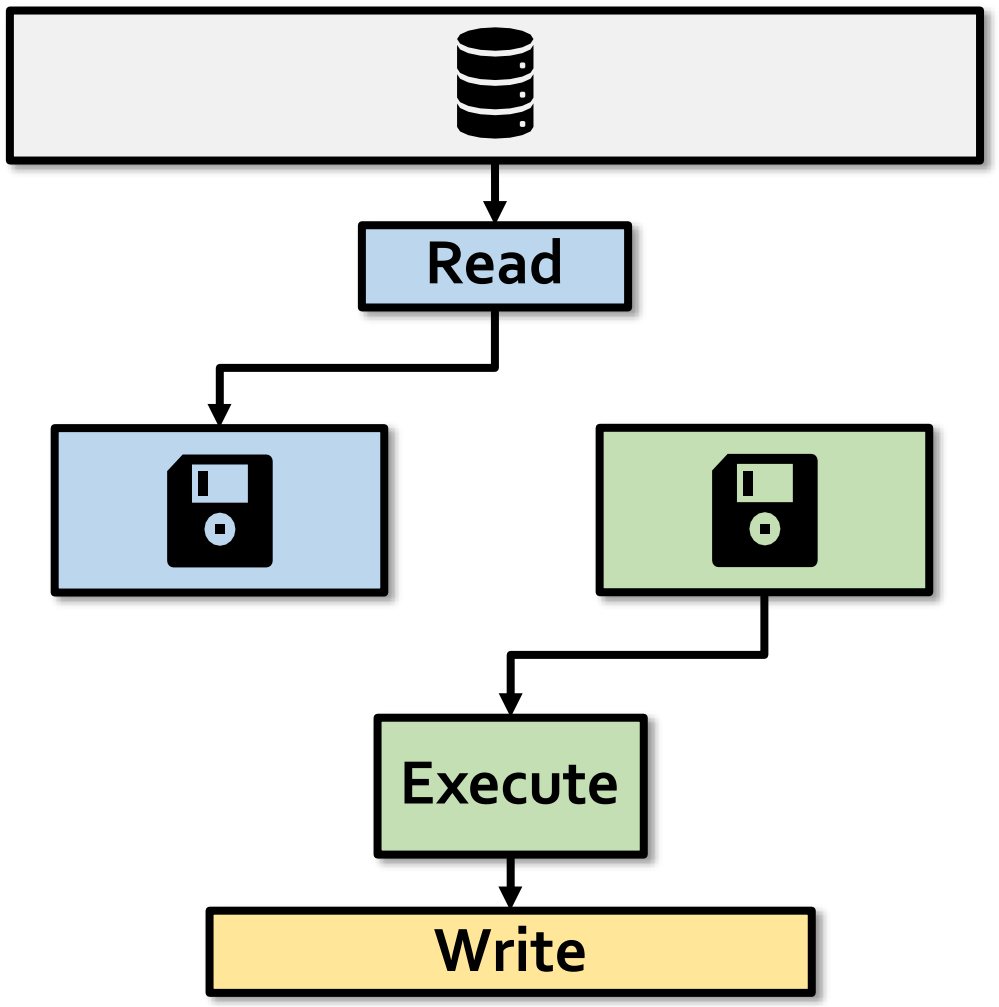

```
read(buf_A, i);
for(int i = 0; i < N; i++) {
    execute(buf_A, buf_B);
    write(buf_B, i);
    if(i < N-1) read(buf_A, i+1);
}
```

How to resolve this dependency?

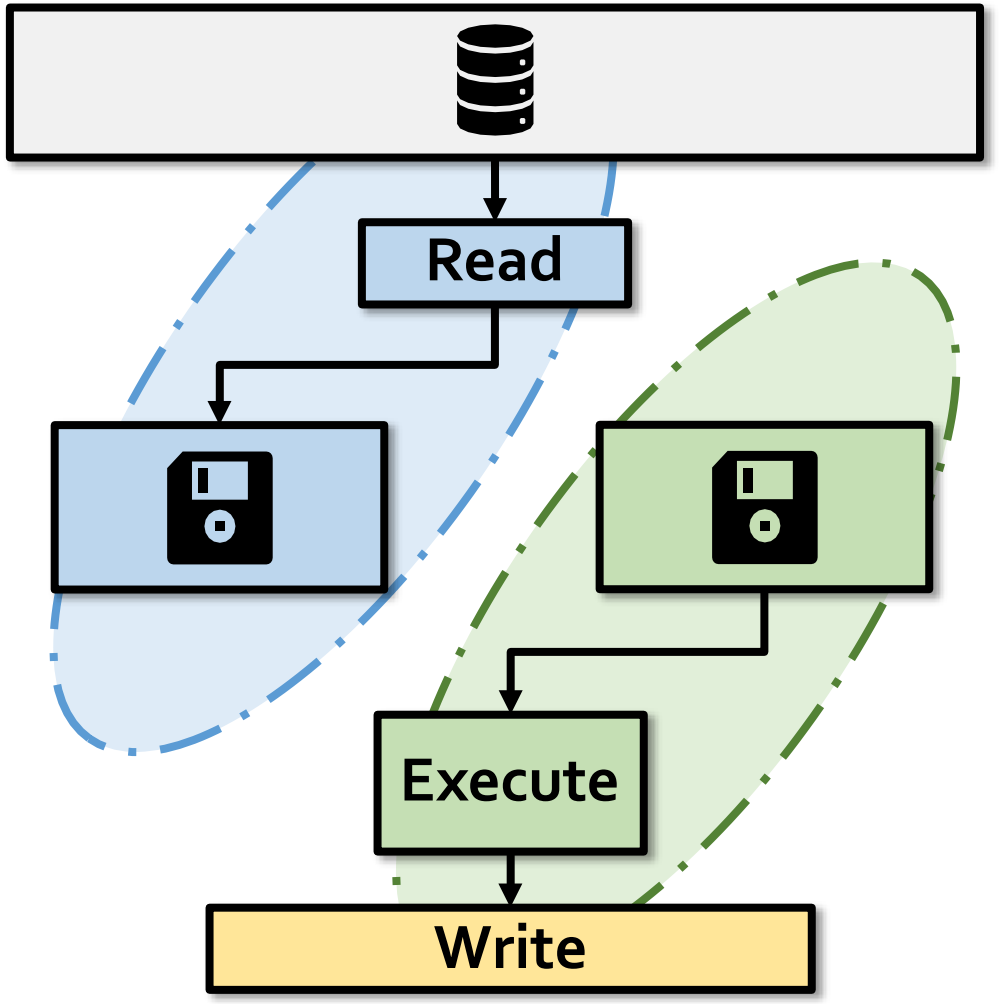
Hidden…



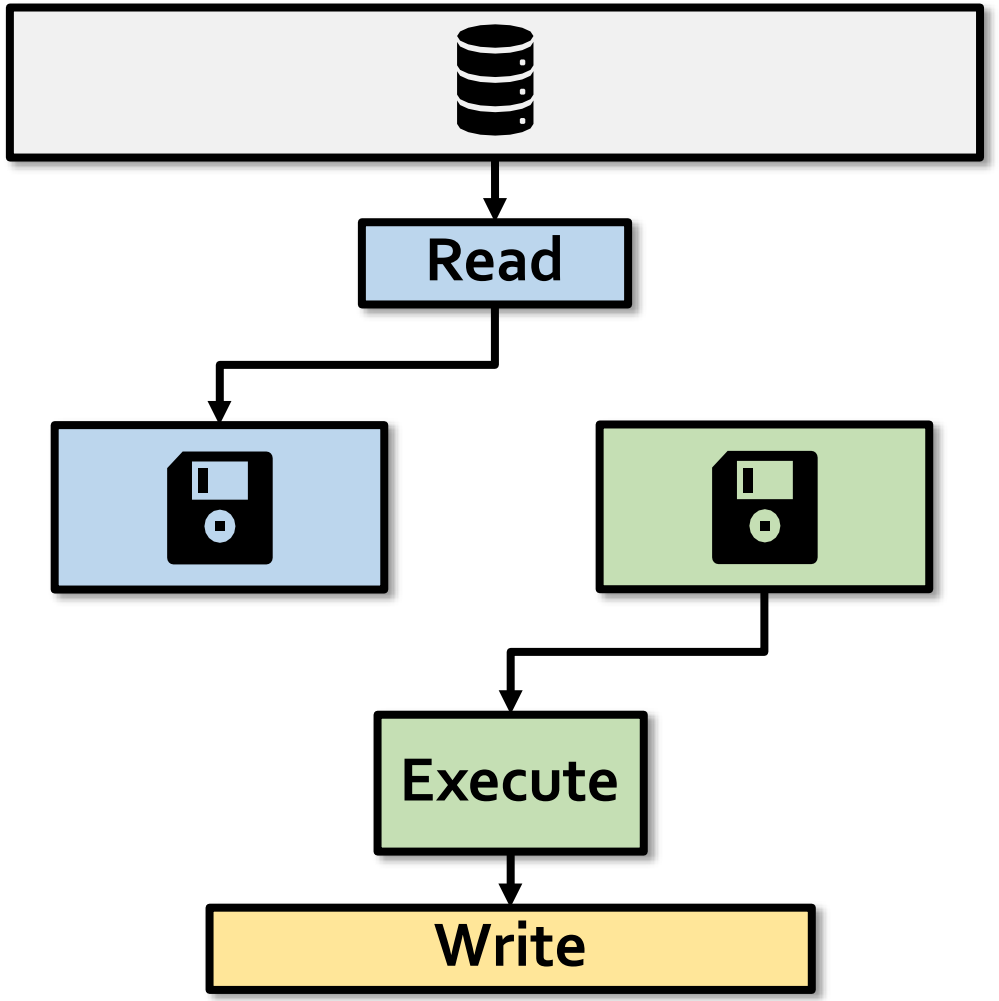
Double Buffer (Ping-Pong Buffer)

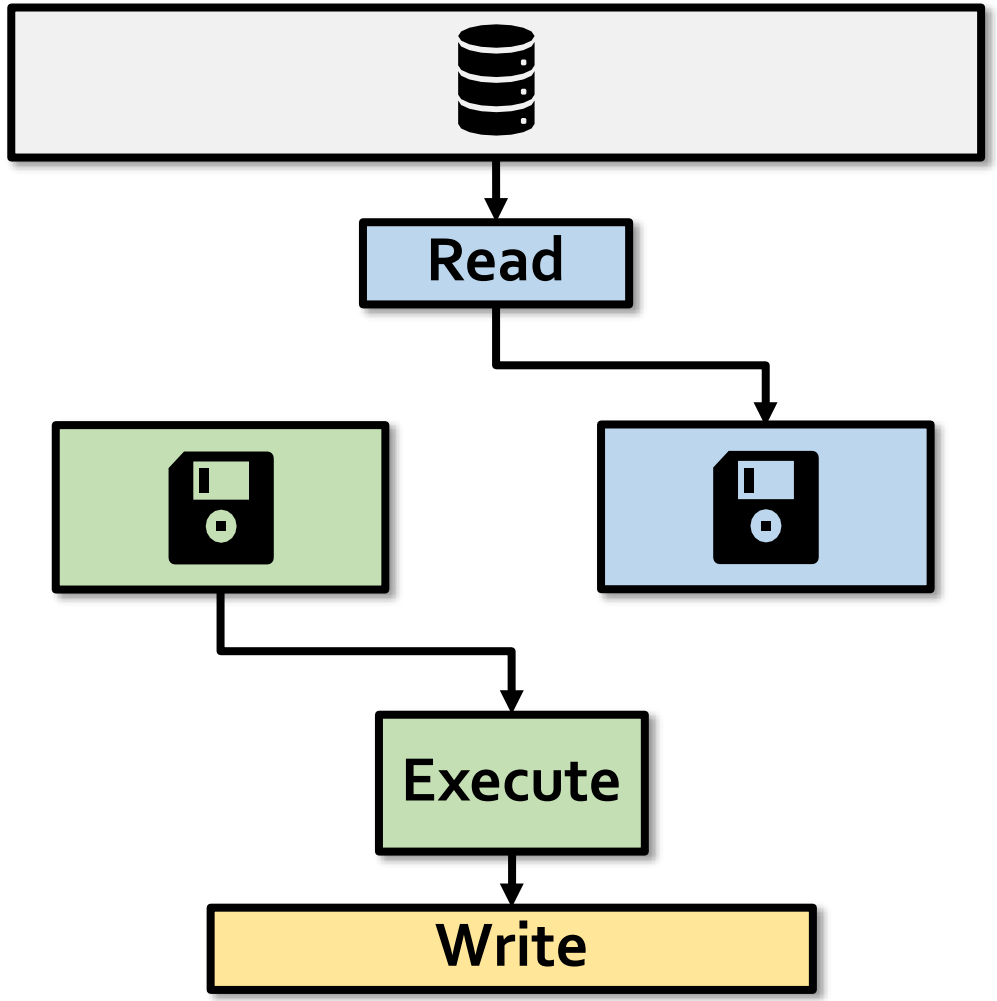


Double Buffer (Ping-Pong Buffer)



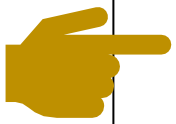
Double Buffer (Ping-Pong Buffer)

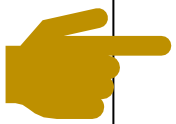


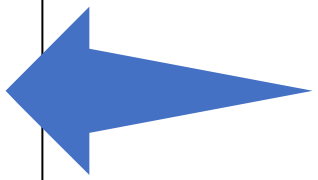


Double Buffer (Ping-Pong Buffer)

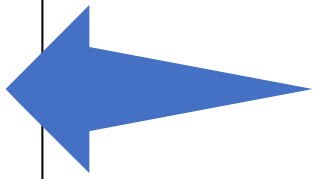
```
read(buf_A_Ping, i);

for(int i = 0; i < N; i++) {
   if( i % 2 == 0 ){
      execute(buf_A_Ping, buf_B);
      write(buf_B, i);
      if( i < N-1 ) read(buf_A_Pong, i+1);
   }
   else if (i % 2 == 1) {
      execute(buf_A_Pong, buf_B);
      write(buf_B, i);
      if( i < N-1 ) read(buf_A_Ping, i+1);
   }
}
```







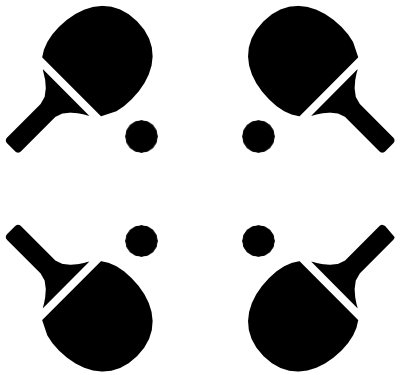
Hidden…



Hidden…

Double Buffer (Ping-Pong Buffer)

```
read(buf_A_Ping, i);
```



```
if( xxx )
write(buf_B_Ping, i);
else
write(buf_B_Pong; i);
```

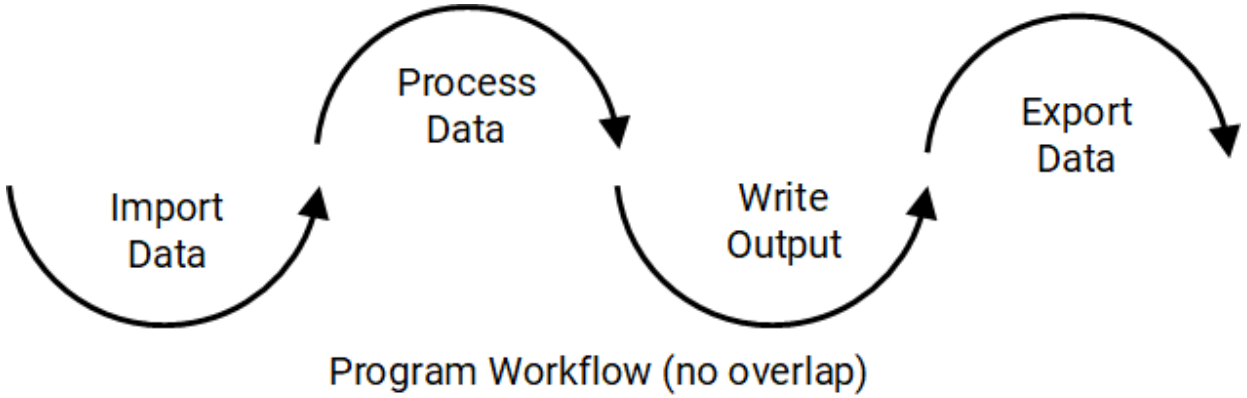
```
for(int i = 0; i < N; i++) {
   if( i % 2 == 0 ){
      execute(buf_A_Ping, buf_B_Ping);
      if( i > 0 ) write(buf_B_Pong, i-1);
      if( i < N-1 )                  i+1);
   }          read(buf_A_Pong,
   else if (i % 2 == 1) {
      execute(buf_A_Pong, buf_B_Pong);
      if( i > 0 ) write(buf_B_Ping, i-1);
      if( i < N-1 ) read(buf_A_Ping, i+1);
   }
}
```

Ping-Pong Buffer Pros and Cons

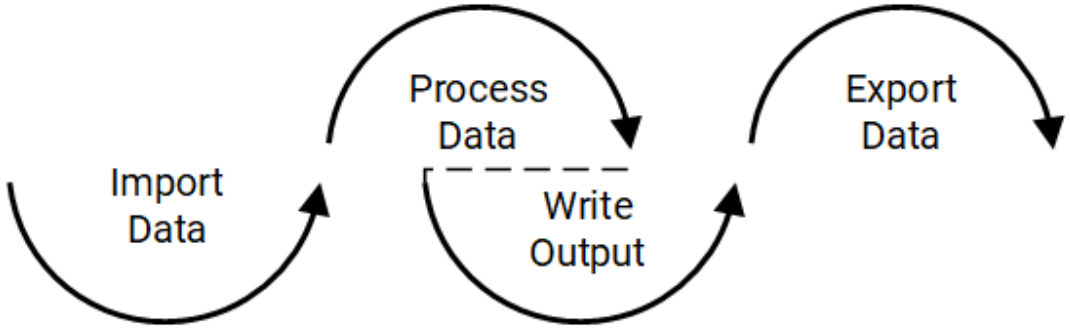
- **Pros**
  - Overlapped execution – shorter latency
- **Cons**
  - Memory overhead (4x)
  - Programmer's effort

Producer-Consumer Paradigm

- **Another way to explain "overlapped execution"**



Program Workflow (no overlap)

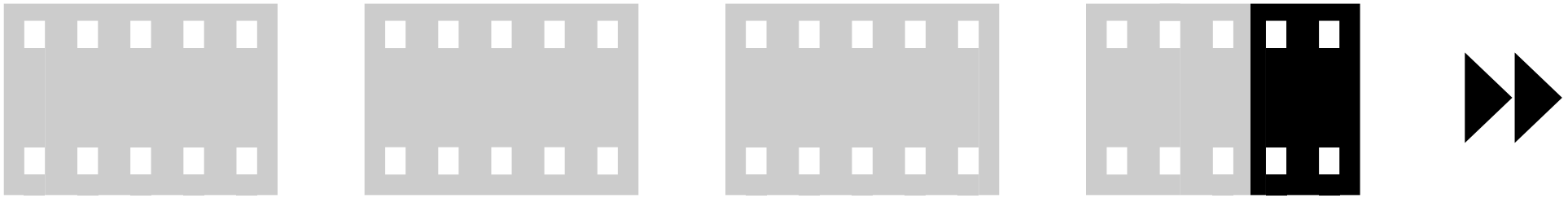


Program Workflow (with overlap)

Data Streaming for Producer-Consumer

- **Stream: an unbounded, continuously updating data set**
  - Unbounded means "of **unknown or of unlimited** size"
  - A sequence of data flowing unidirectionally between a source (producer) process and a destination (consumer) process

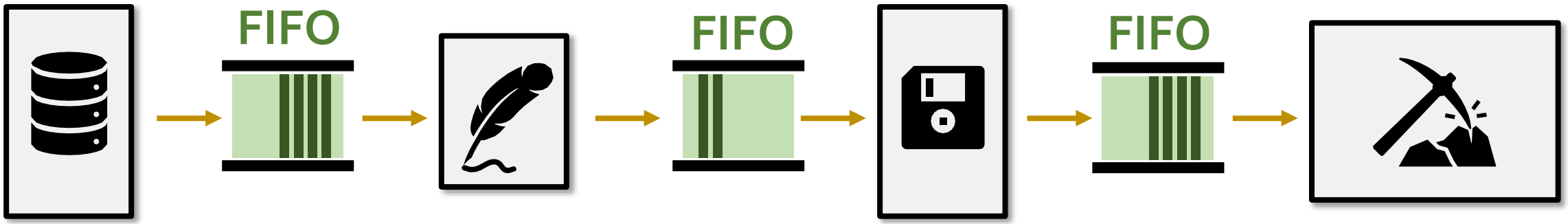
- **Example: real-time video, audio, etc.**



Started processing – don't wait for the entire

Data Streaming for Producer-Consumer

- **Enabled by FIFO (first-in first-out) buffers**
  - The consumer process can start accessing the data inside the FIFO buffer as soon as the producer inserts the data into the buffer
  - If the buffer is full/empty, automatically stalls



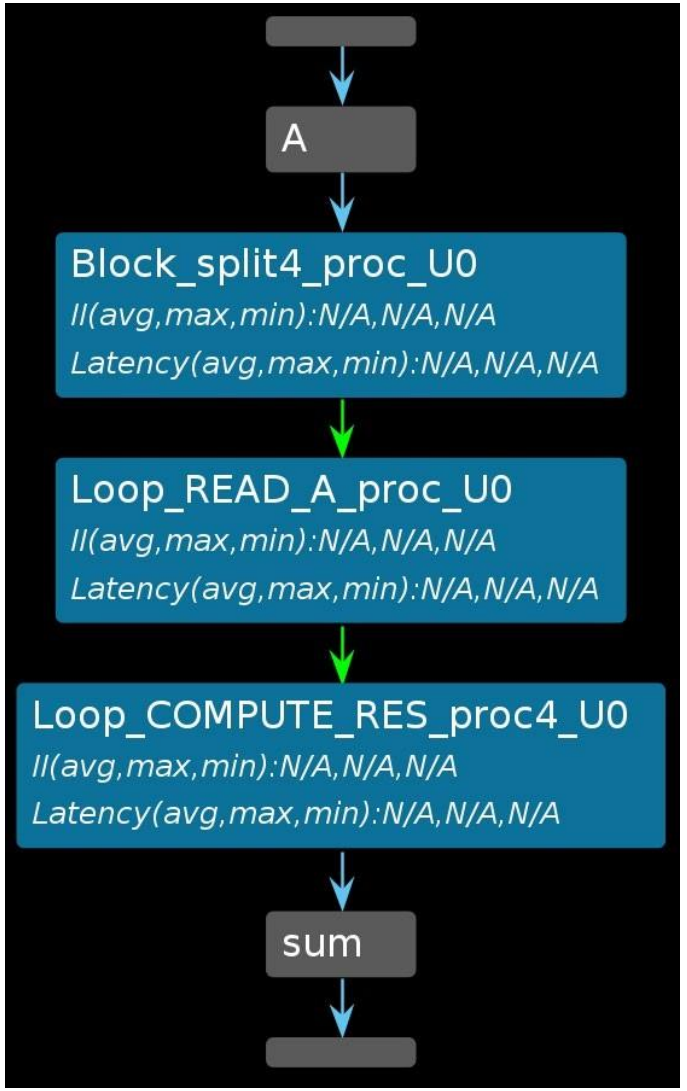
Data Streaming in HLS

- **Step 1: create FIFOs using hls::stream<type>**
  - Specify a depth (how large the FIFO is)
- **Step 2: organize into two functions or loops**
  - One writing to FIFO, one reading from FIFO
- **Step 3: apply dataflow pragma**

Data Streaming in HLS – Example

```
void test( FIX_TYPE* A, FIX_TYPE* sum ) {

#pragma HLS dataflow

    hls::stream<FIX_TYPE> buffer;
#pragma HLS STREAM variable=buffer depth=10


    READ_A: for(int i = 0; i < 100; i++) {
        buffer.write(A[i]);
    }

    COMPUTE_RES: for(int i = 0; i < 100; i++) {
        FIX_TYPE d = buffer.read();
        FIX_TYPE res = d * d + i;
        sum[i] = res;

    }
}
```

1

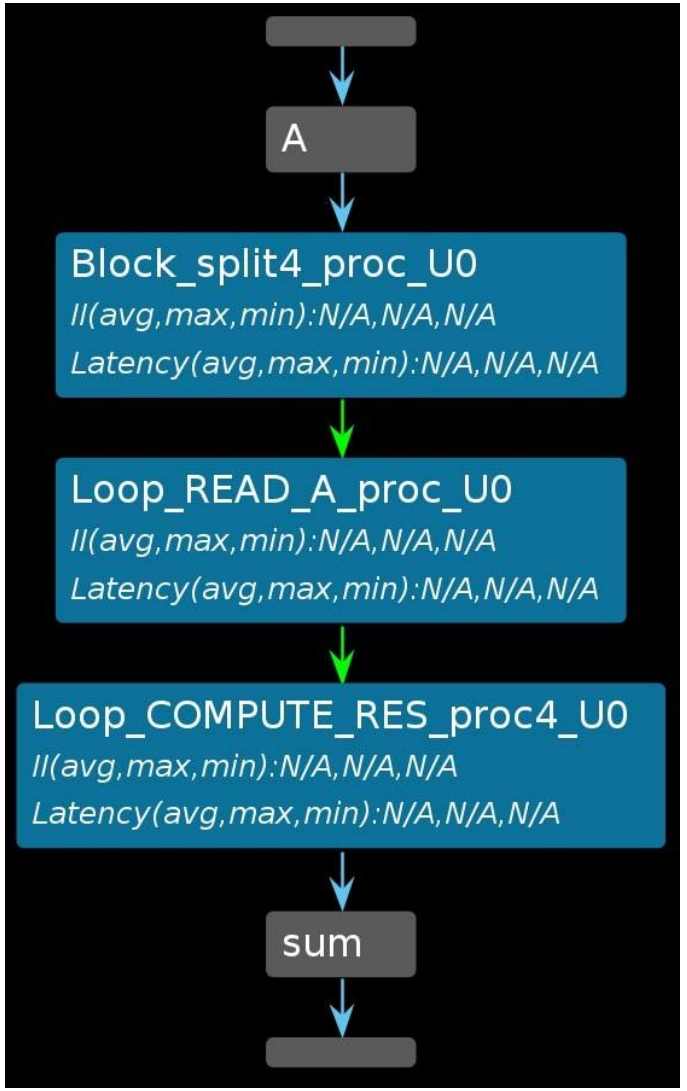


Data Streaming in HLS – Example

```
void test( FIX_TYPE* A, FIX_TYPE* sum ) {

#pragma HLS dataflow

    hls::stream<FIX_TYPE> buffer;
#pragma HLS STREAM variable=buffer depth=10


    READ_A: for(int i = 0; i < 100; i++) {
        buffer.write(A[i]);
    }

    COMPUTE_RES: for(int i = 0; i < 100; i++) {
        FIX_TYPE d = buffer.read();
        FIX_TYPE res = d * d + i;
        sum[i] = res;

    }
}
```

2

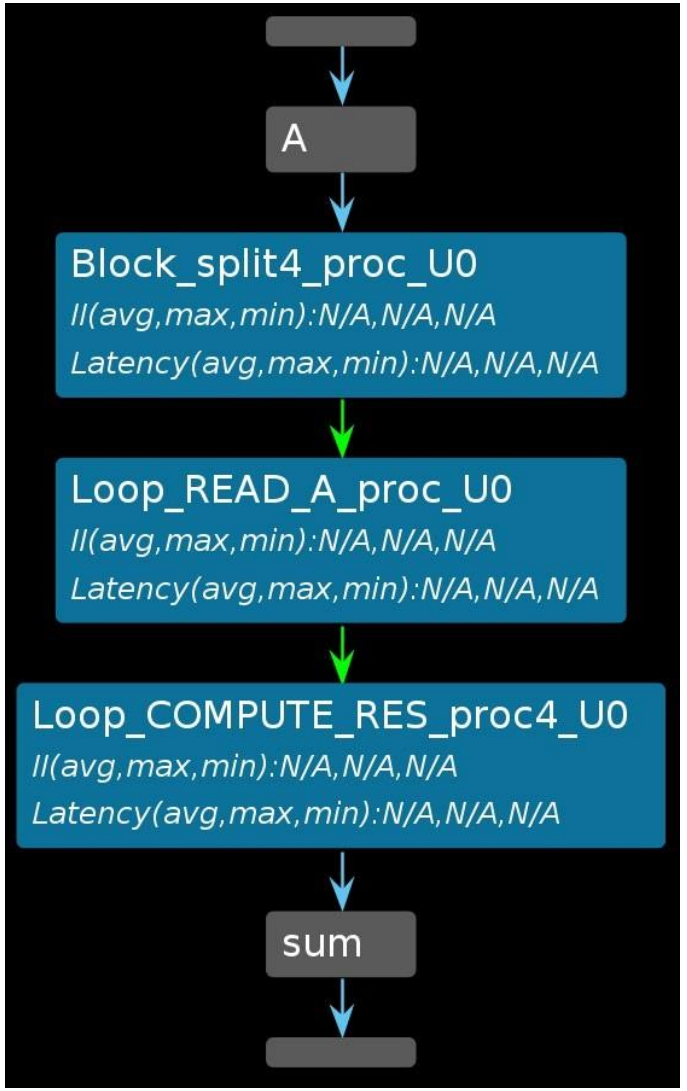
2



Data Streaming in HLS – Example

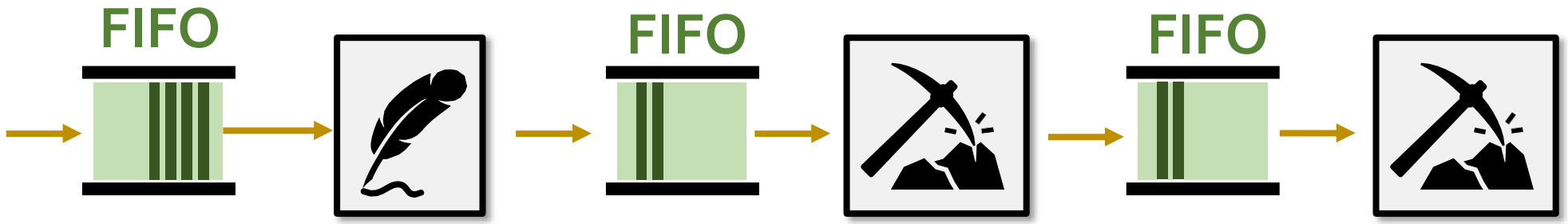
3

```
void test( FIX_TYPE* A, FIX_TYPE* sum ) {

#pragma HLS dataflow

    hls::stream<FIX_TYPE> buffer;
#pragma HLS STREAM variable=buffer depth=10


    READ_A: for(int i = 0; i < 100; i++) {
        buffer.write(A[i]);
    }

    COMPUTE_RES: for(int i = 0; i < 100; i++) {
        FIX_TYPE d = buffer.read();
        FIX_TYPE res = d * d + i;
        sum[i] = res;

    }
}
```



Pitfalls of Dataflow and Streaming

- Single Producer, Single Consumer – everything must be streamlined, no bypass
- No Feedback between tasks
- No Conditional execution of tasks
- No Loops with multiple exit conditions

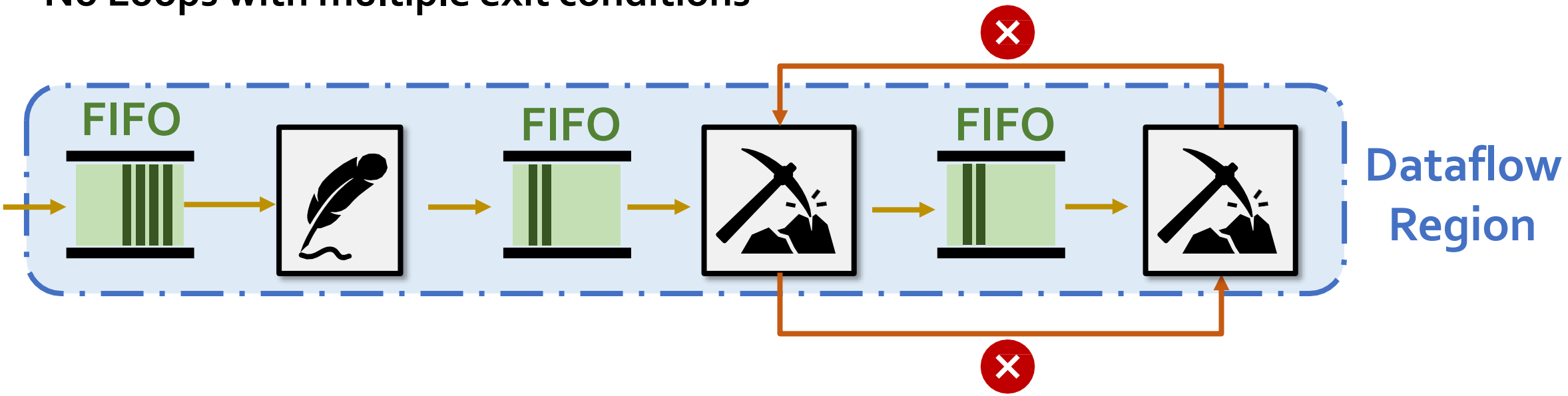


Pitfalls of Dataflow and Streaming

- Single Producer, Single Consumer – everything must be streamlined, no bypass
- No Feedback between tasks
- No Conditional execution of tasks
- No Loops with multiple exit conditions

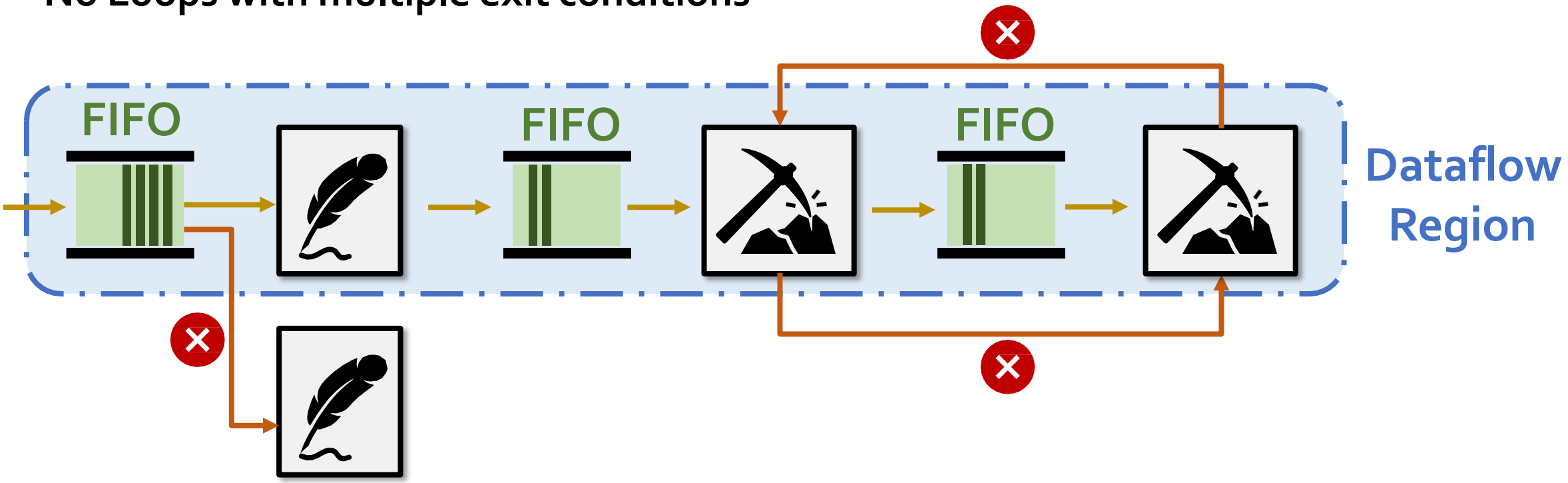
Pitfalls of Dataflow and Streaming

- Single Producer, Single Consumer – everything must be streamlined, no bypass
- No Feedback between tasks
- No Conditional execution of tasks
- No Loops with multiple exit conditions



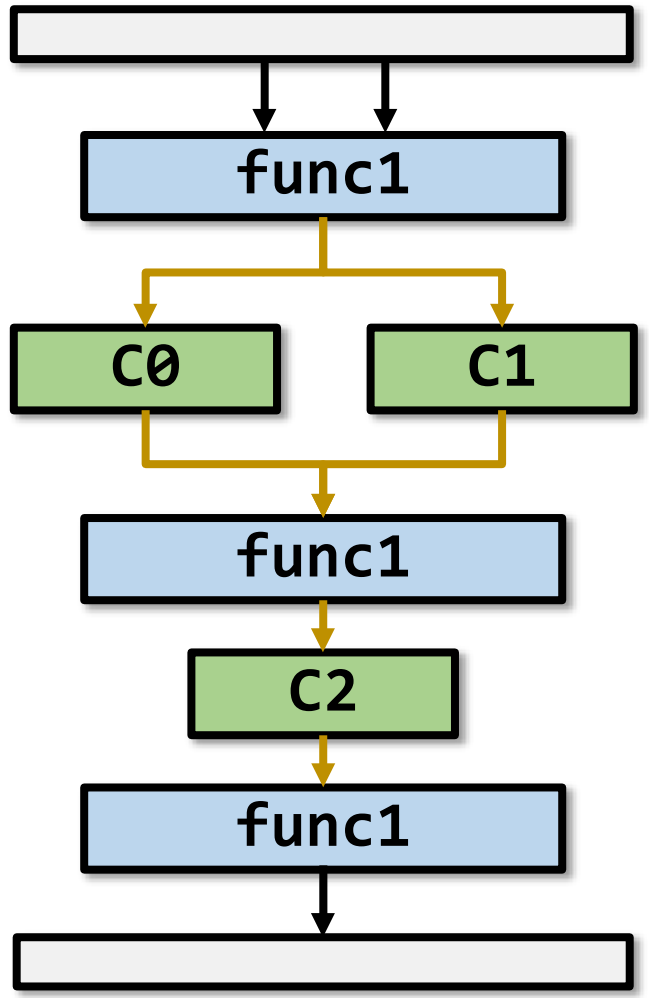
Pitfalls of Dataflow and Streaming

- Single Producer, Single Consumer – everything must be streamlined, no bypass
- No Feedback between tasks
- No Conditional execution of tasks
- No Loops with multiple exit conditions



Canonical Forms

```
void dataflow_top(Input0, Input1, Output0, Output1)
{
    for (int i = 0; i < N; i++) {
#pragma HLS dataflow

    Streaming_Buffer C0, C1, C2;

    func1(read_Input0, read_Input1, write_C0, write_C1);

    func2(read_C0, read C1, write_C2);

    func3(read_C2, write_Output0, write_Output1);

    }
}
```



Synthesis v.s. Implementation

- This is what you get from Synthesis
- Performance and Resource Estimation

Utilization Estimates

Summary

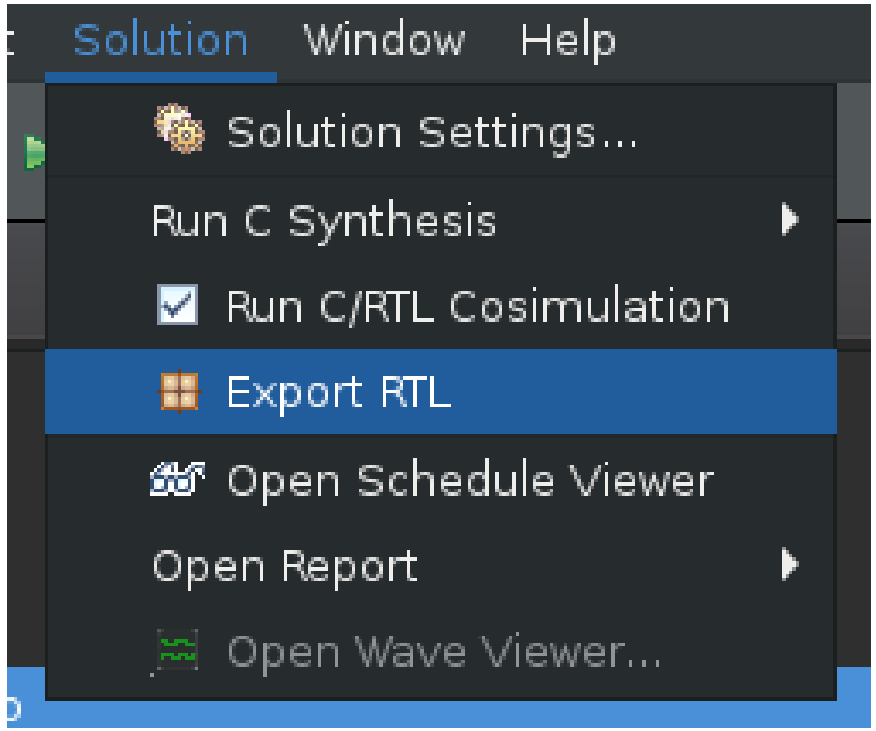
Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	1	-	-	-
Expression	-	-	0	414	-
FIFO	-	-	-	-	-
Instance	6	0	2433	3248	-
Memory	32	-	0	0	-
Multiplexer	-	-	-	605	-
Register	-	-	969	160	-
Total	38	1	3402	4427	0
Available	40	40	16000	8000	0
Utilization (%)	95	2	21	55	0

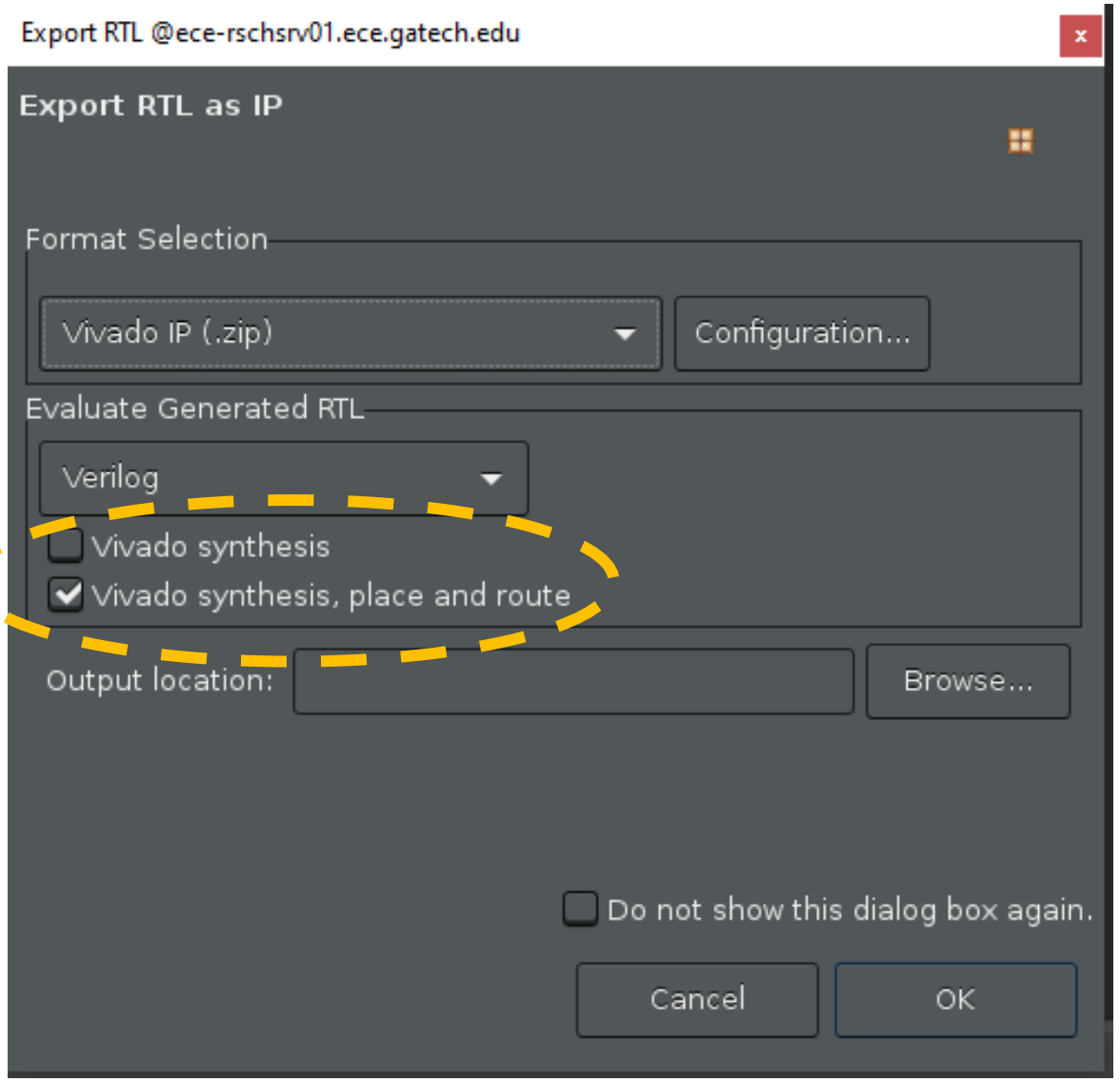
Performance & Resource Estimates

Modules Loops

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
test		-	31236	3.120E5	-	31237	-	no	38	1	3402	4427	0
fooA		-	10017	1.000E5	-	10017	-	no	0	0	329	511	0
fooB		-	5017	5.017E4	-	5017	-	no	0	0	328	510	0
L1_VITIS_LOOP_420_1		-	10009	1.000E5	11	1	10000	yes	-	-	-	-	-
L2		-	11200	1.120E5	112	-	100	no	-	-	-	-	-

Synthesis v.s. Implementation





- **To run Implementation**
- **If run into errors:**
  - Try "lastyear vitis_hls"

Synthesis v.s. Implementation

Synthesis

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	7.300 ns	2.70 ns

Utilization Estimates

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	1	-	-	-
Expression	-	-	0	414	-
FIFO	-	-	-	-	-
Instance	6	0	2433	3248	-
Memory	32	-	0	0	-
Multiplexer	-	-	-	605	-
Register	-	-	969	160	-
Total	38	1	3402	4427	0
Available	40	40	16000	8000	0
Utilization (%)	95	2	21	55	0

Implementation

Resource Usage

	Verilog
SLICE	1510
LUT	3220
FF	6109
DSP	4
BRAM	38
SRL	303

Final Timing

	Verilog
CP required	10.000
CP achieved post-synthesis	7.820
CP achieved post-implementation	8.759

Timing met

Summary

- Widening the memory port
- Double buffer to hide the data loading latency
- More advanced (challenging) technique: streaming and dataflow
  - BUT! Dataflow architecture is ***really*** efficient and will be very promising in the future!