



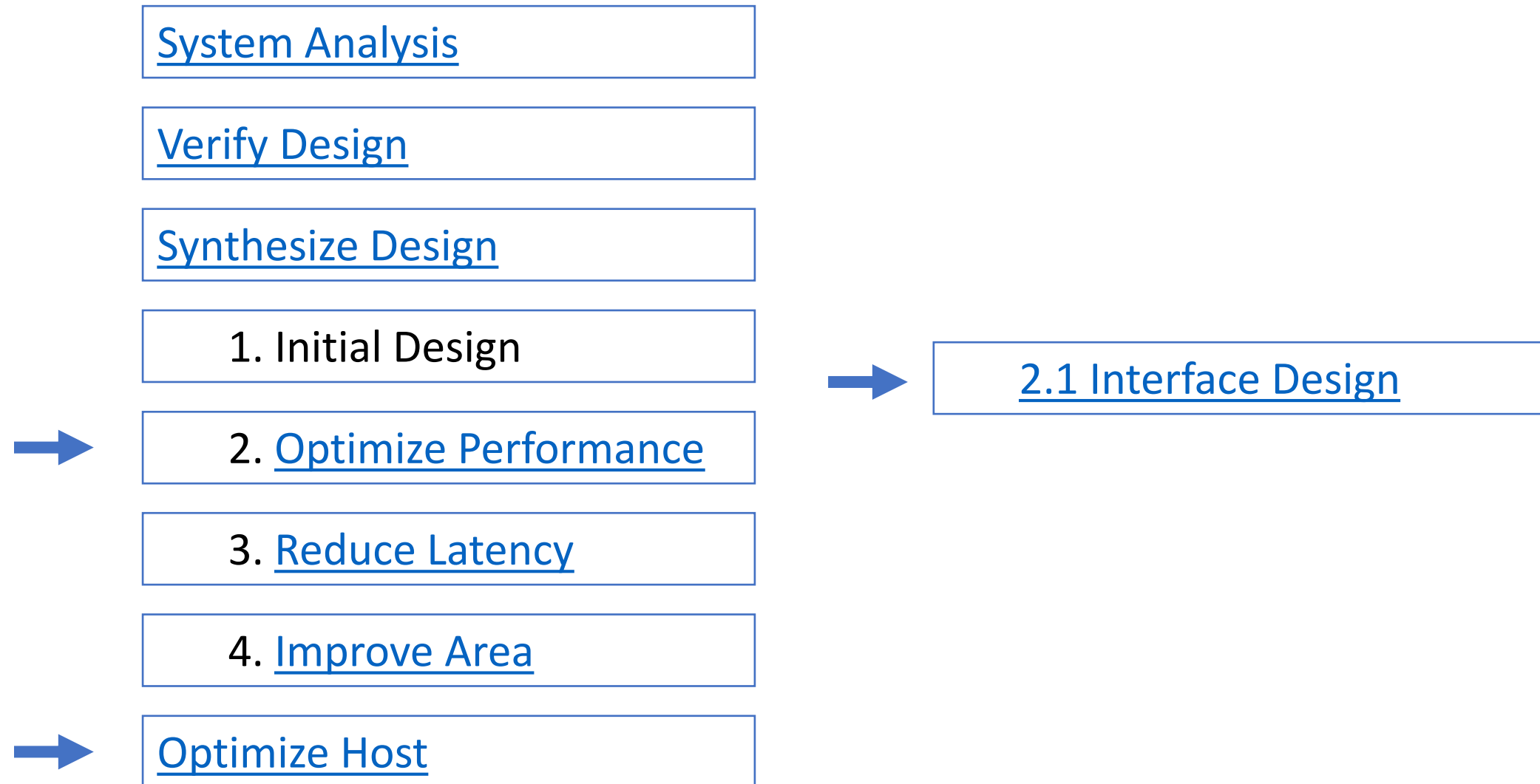
Bridge of Life  
Education

# Advanced SOC Design

Application Acceleration Best Practice

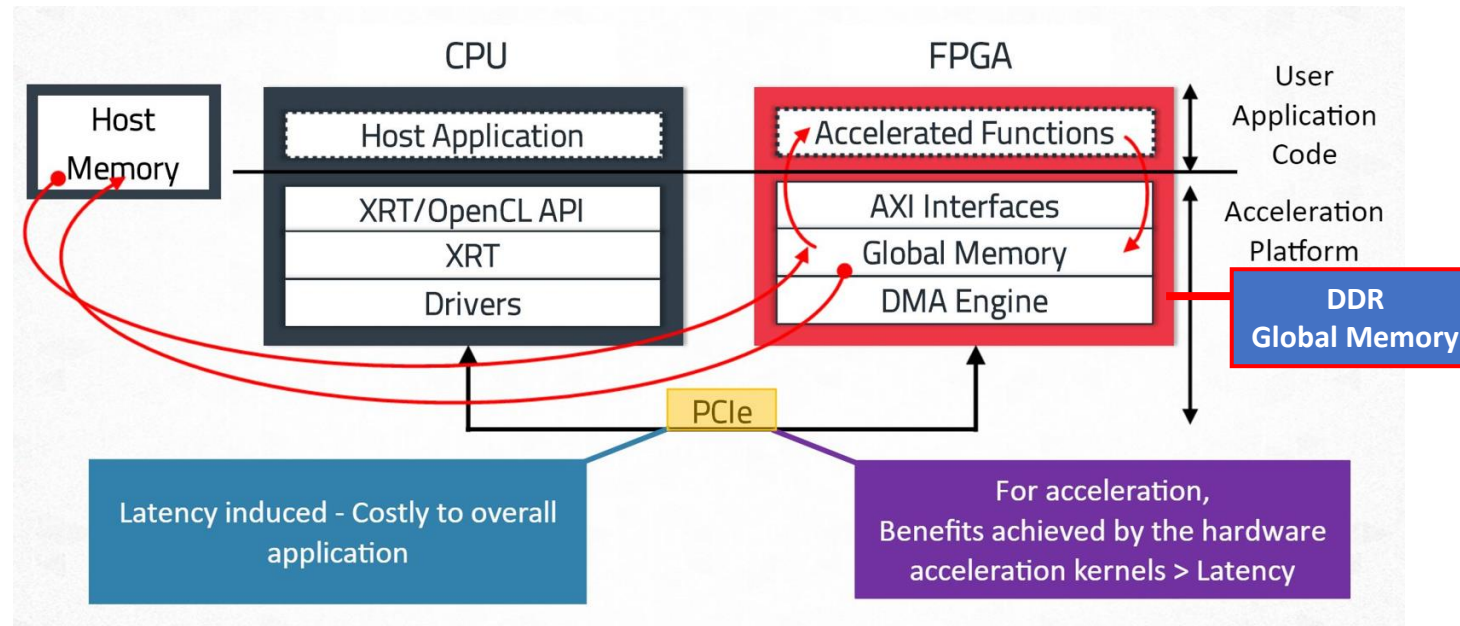
Jiin Lai

# Steps of System Design for Application Acceleration



# System Analysis

# Vitis Programming and Execution Model

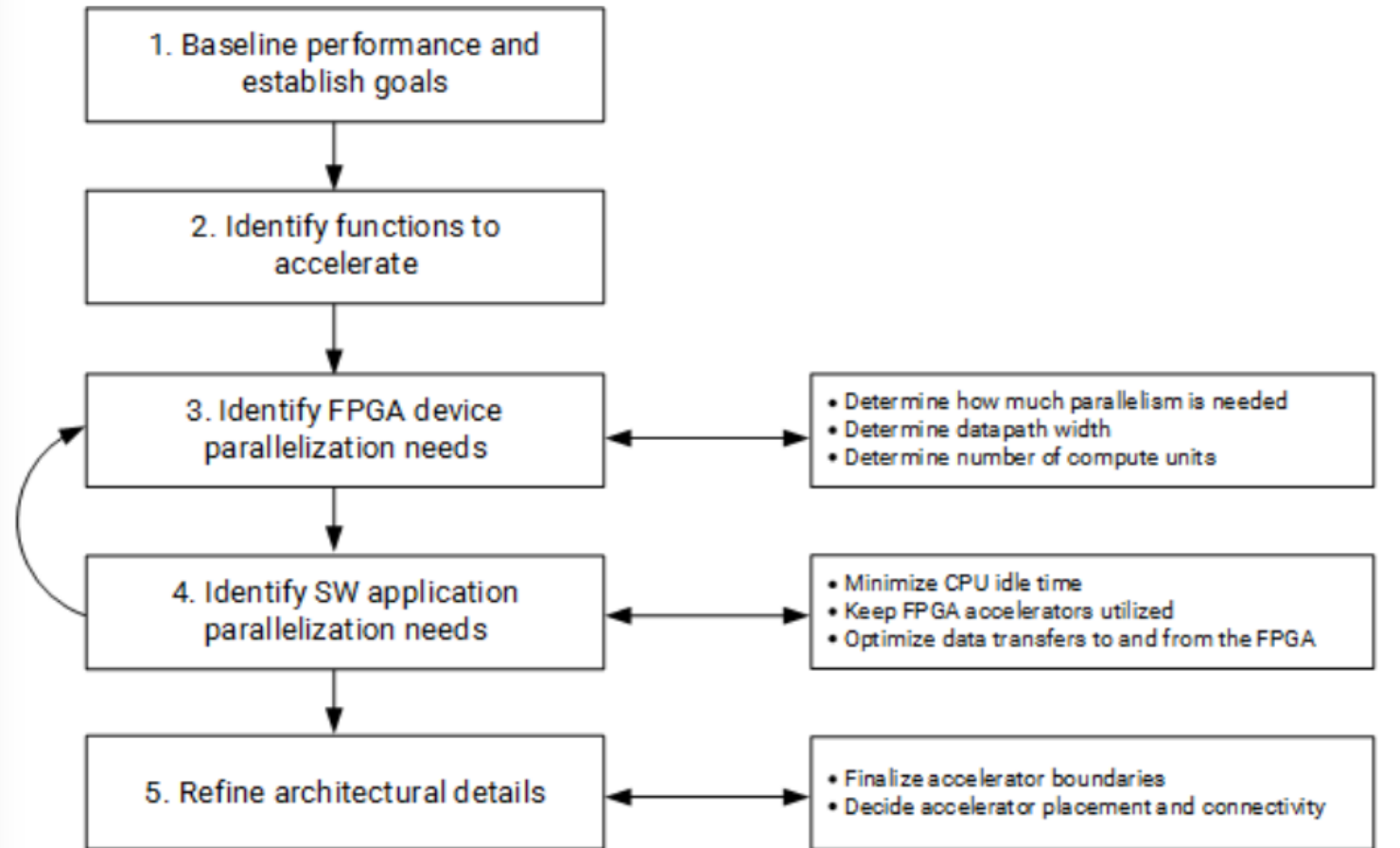


1. The host program transfers the data needed by a kernel into the global memory of the FPGA device.
2. The host program sets up the input parameters of the kernel.
3. The host program triggers the execution of the kernel.
4. The kernel performs the required computation, accessing global memory to read or write data as necessary. Kernels can also use streaming connections to communicate with other kernels.
5. The kernel notifies the host that it has completed its task.
6. The host program can transfer the data from global memory to host memory or give ownership of the data to another kernel.

Kernel Direct Access Host Memory: <https://xilinx.github.io/XRT/master/html/hm.html>

# Optimization Guideline

Figure: Methodology for Architecting the Application



X23282-092619

# General Guideline on System Performance

- Think of Throughput, not just Latency
- Amdahl's Law
  - Fraction  $f$  is parallelizable by a factor  $p$   
then  $\text{speedup} = 1 / ((1-f) + f/p) \sim 1/(1-f)$
  - If only 50% can be accelerated, maximum 2x speedup
- Target accelerated function
  - Large compute bound, not memory bound
  - Do not require frequent synchronization with host
- Consider end-to-end system overhead (I/O, Scheduling ... )

System Analysis

Verify Design

Synthesize Design

1. Initial Design

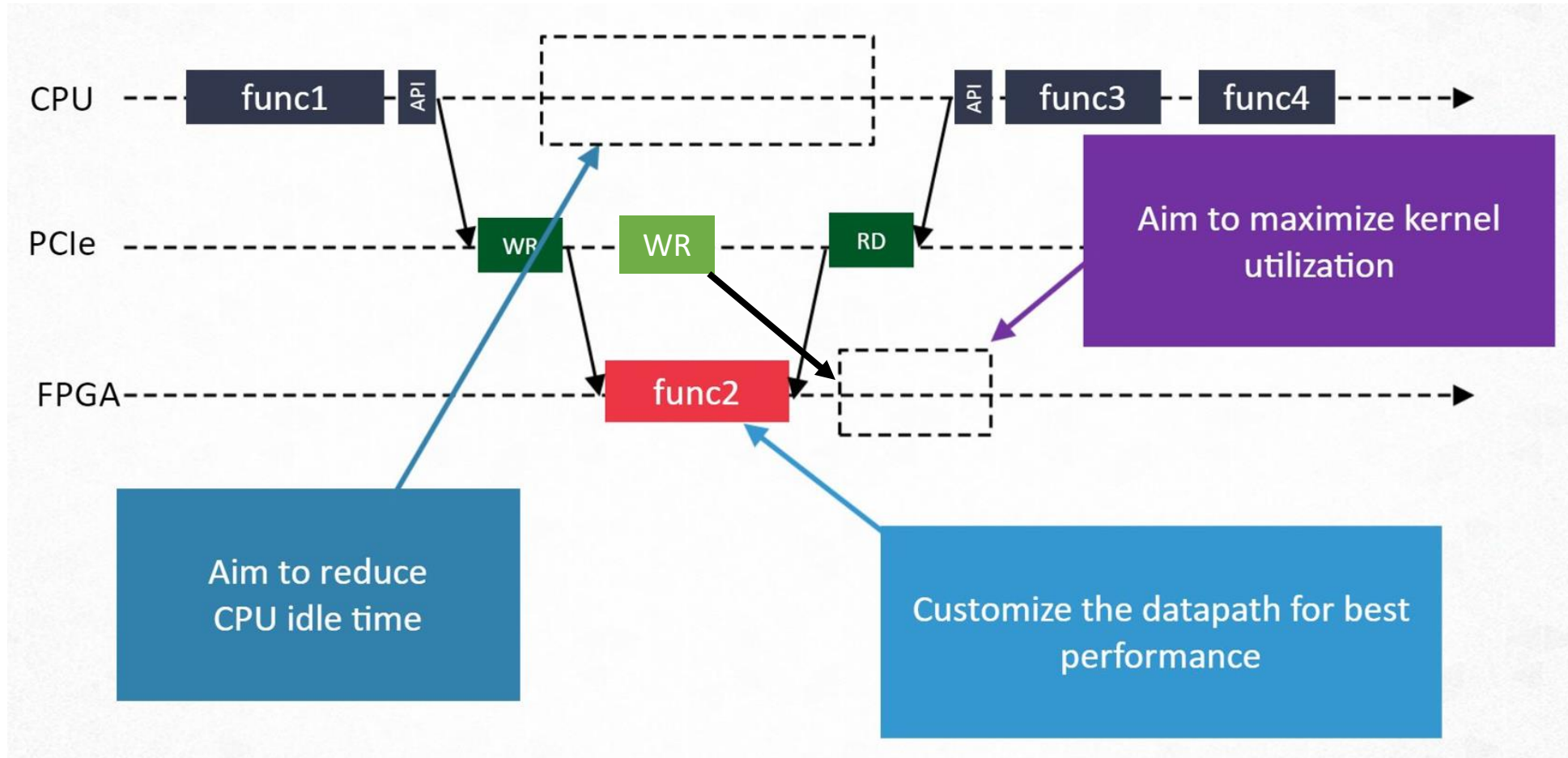
2. Optimize Performance

3. Reduce Latency

4. Improve Area

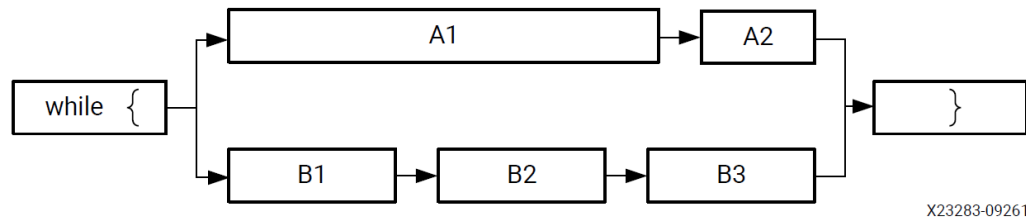
Optimize Host

# Area to improve Throughput



# Identify Functions to Accelerate

- Performance Bottlenecks, consider the entire application's performance



- Identify Acceleration Potential
  - What is the parallelism of the function? Task-level, data-level
  - Computational Intensity? #-of-op / #-of-words-moved-from-slow-memory
  - Data access locality ? Spatial / Temporal locality
  - Bottleneck come from I/O, computation, data movement or synchronization ?
    - PCIe throughput of 10GBs, software throughput of 50MBs, acceleration factor?



# Pay attention to the whole system workflow

- Add storage & network into the picture
- Data transfer among Network – Storage – CPU – Accelerator
  - Use Peer-to-Peer transfer if possible (NVMe-TCP)
  - Data format mismatch
  - Data scattered
- Accelerator / DMA setup
- Event Synchronization
- Consider
  - Compute on the fly. Dataflow v.s. Store-and-Forward.
  - Pipeline - hiding the overhead

# Networked Computational Storage Explained

## Peer-to-peer transfer & Compute on-the-fly

- Conventional network storage v.s. NVMe-TCP

# Verify Design

System Analysis

Verify Design

Synthesize Design

1. Initial Design

2. Optimize Performance

3. Reduce Latency

4. Improve Area

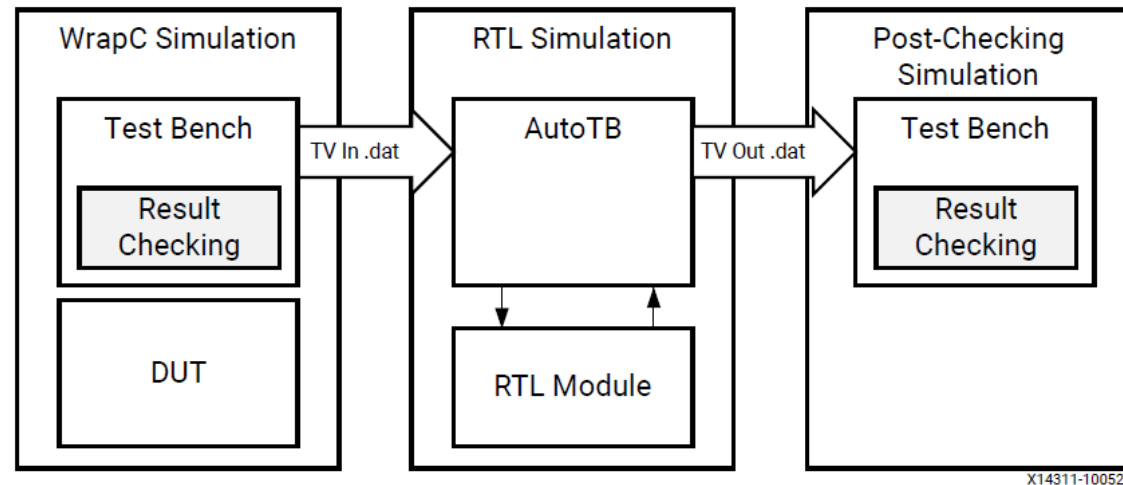
Optimize Host

# Guideline for Writing a Test Bench

- Separate the kernel function from the test bench and header files.
- **System-level integrated test** is imperative.
  - Can test bench be integrated into application body?
- Ensure all aspects of the function is fully verified.
  - Check code coverage, e.g. fifo empty/full, random IO latency, all branch condition ...
- Use the same test bench for C/RTL Co-simulation.
  - Verify kernel performance, i.e. II and latency.
  - Ensure test bench exercise corner cases.
- **Auto-check for correctness** in test bench, including run time log file. The return value of function main() indicates the following:
  - Zero: Results are correct.
  - Non-zero value: Results are incorrect. The non-zero results indicate the type of failure.
- **Embed debugging fixture** in the system workflow with minimum overhead
  - Hashing/signature

# C/RTL Co-simulation

1. Phase#1: C simulation is executed to prepare the “input vectors” to the top-level function.
2. Phase#2: RTL simulation starts; it takes the “input vectors” and generates the “output vectors”
3. Phase#3: Return to main() function. Compare the “output vectors” with expected data.
4. Provide sufficient FIFO depth to avoid stall.
5. Check for unsupported optimizations for Co-Simulation, e.g. reshape on array argument
6. Use Waveform viewer or Co-simulation deadlock viewer if deadlock happens



# Synthesize Design

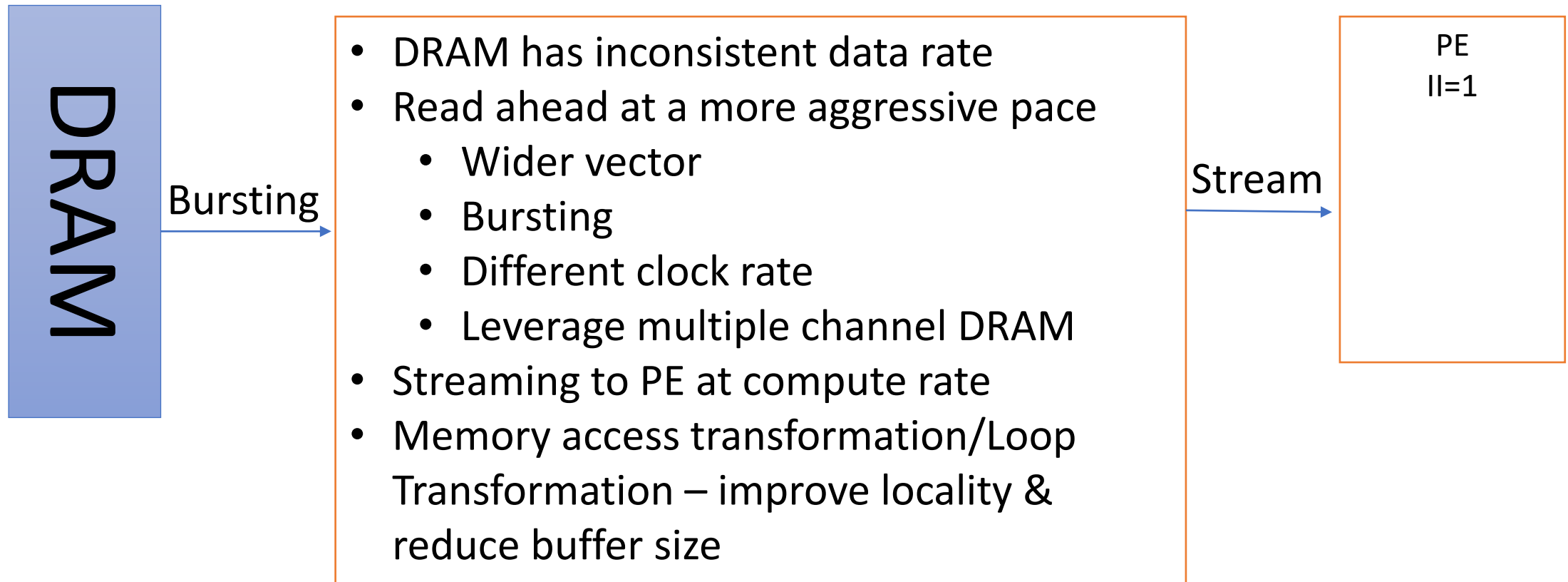
- Use directive or pragma to guide HLS optimization process (try to minimize the restructure the code to meet performance)
- Study Synthesis Summary Report
- Use Schedule Viewer, Property viewer, Dataflow viewer to analyze datapath and control flow
- Review Synthesis Logfile - synthesis steps, checking status, optimization result

Synthesis log messages contains

- Project and solution initialization loads source and constraints files, and configures the active solution for synthesis.
- Start compilation reads source files into memory.
- Interface detection and setup reviews and generates port and block interfaces for the function.
- Burst read and write analysis for ports/interfaces.
- Compiler transforms code to operations.
- Performs Synthesizability checks.
- Automatic pipelining of loops at trip-count threshold.
- Unrolling loops, both automatic and user-directed.
- Balance expressions using associative and commutative properties.
- Loop flattening to reduce loop hierarchy.
- Partial write detection (writing part of a memory word)
- Finish architecture synthesis, start scheduling.
- End scheduling, generate RTL code.
- Report F-Max and loop constraint status.

# Interface Best Practice

# Memory Architecture





# Interface Best Practice

- **Separate IO accesses from compute**
  - Apply different optimization strategy for IO and compute
- **Burst Transfer**
  - Need local buffer
  - Separate concurrent access memory port
- **Pointer**
  - Avoid pointer aliasing – Compiler can not pipeline the loop
- **Vector data type** (increase width, saved data in local memory)
- **Streaming with vector** (aggregate struct, increase width, no data buffer )

```
#include <hls_vector.h>
using vint16 = hls::vector<int, 16>;
```

```
void wide_vadd(vint16 *a, vint16 *b, vint16 *c, int size) {
#pragma HLS INTERFACE m_axi bundle=in_a port=a depth=1024
#pragma HLS INTERFACE m_axi bundle=in_b port=b depth=1024
#pragma HLS INTERFACE m_axi bundle=out_c port=c depth=1024
int i, buf_a[N], buf_b[N], buf_c[N];
```

```
// memcpy – burst transfer
memcpy(buf_a, (const vint16*) a, 50*sizeof(int));
memcpy(buf_b, (const vint16*) b, 50*sizeof(int));
```

```
for (int i = 0; i < (size/16); i++) {
    buf_c[i] = a[i] + b[i]; }
```

```
// for loop + PIPELINE
for(i=0; i < N; i++){
    #pragma HLS PIPELINE
    c[i] = buf_c[i]; }
}
```

# Pointer Interface

- The pointer on the interface is read or written once per function call
- Avoid Multiple access - using hls::stream
  - Declared as volatile
  - Perform co-simulation
- Avoid pointer arithmetic, use array with index

```
#include "pointer_arith.h"

void pointer_arith (dio_t *d) {
    static int acc = 0;
    int i;

    for (i=0;i<4;i++) {
        acc += *(d+i+1); // start from 2nd value
        *(d+i) = acc;
    }
}
```



```
#include "array_arith.h"
void array_arith (dio_t d[5]) {
    static int acc = 0;
    int i;
    for (i=0;i<4;i++) {
        acc += d[i+1];
        d[i] = acc;
    }
}
```

```
#include "pointer_stream_better.h"

void pointer_stream_better ( volatile dout_t
*d_o, volatile din_t *d_i) {
    din_t acc = 0;

    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
    acc += *d_i;
    acc += *d_i;
    *d_o = acc;
}
```

# Memory Interface

- Use multiple memory ports – bundle
- Increase memory port width to 512 bit
  - Use vector for SIMD – data parallelism
- Minimize read/write
- Read/write by burst
  - Use PIPELINE
  - No conditional statement
- Read data once and use a local cache for data reused.

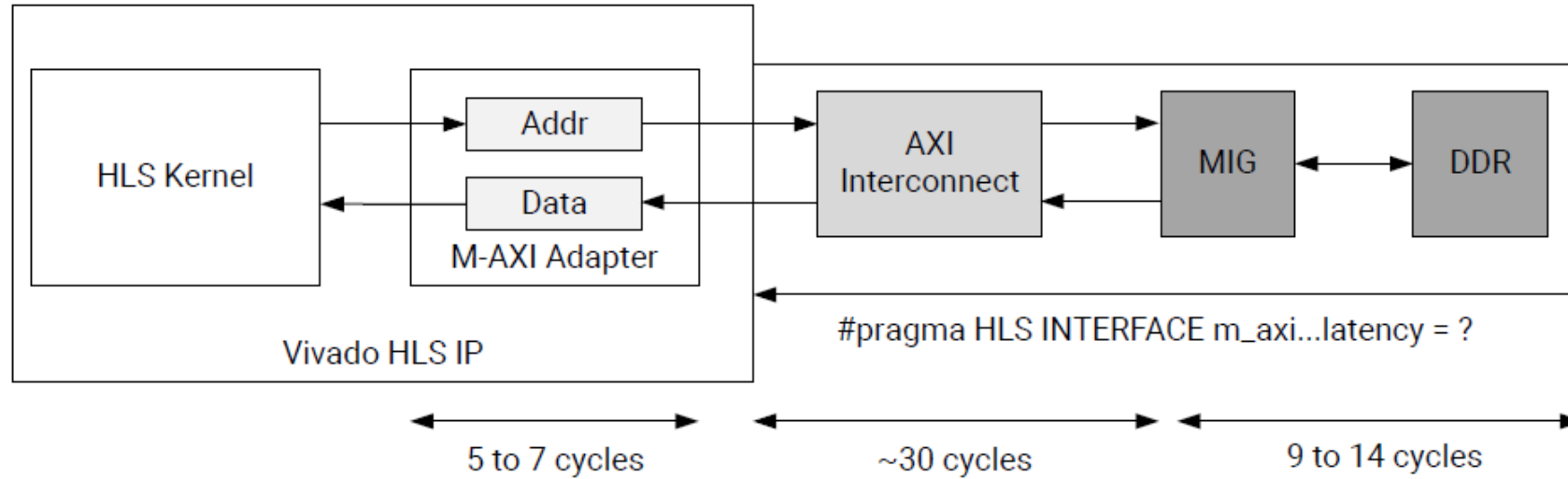
```
#include <hls_vector.h>
using vint16 = hls::vector<int, 16>;
```

```
void wide_vadd(vint16 *a, vint16 *b, vint16 *c, int size) {
    #pragma HLS INTERFACE m_axi bundle=in_a port=a depth=1024
    #pragma HLS INTERFACE m_axi bundle=in_b port=b depth=1024
    #pragma HLS INTERFACE m_axi bundle=out_c port=c depth=1024

    for (int i = 0; i < (N/16); i++) {
        c[i] = a[i] + b[i]; }
    } }
```

```
hls::stream<datatype_t> str;
INPUT_READ: for(int i=0; i<INPUT_SIZE; i++) {
    #pragma HLS PIPELINE
    str.write(inp[i]); // Reading from Input interface
}
```

# Latency & Bus utilization



- **depth:** Specifies the maximum number of samples for the test bench to process.
- **latency:** Specifies the expected latency of AXI4 interface.
- **max\_read\_burst\_length, max\_write\_burst\_length:** Specifies the maximum number of data values read during a burst transfer.
- **num\_read\_outstanding, num\_write\_outstanding:** Specifies how many read requests can be made to the AXI4 bus without a response before the design stalls.

It needs storage to hold `num_read_outstanding` requests, and

and **data FIFO size = `num_read_outstanding * max_read_burst_length * word_size`.**

# Interface Best Practice - Streaming

- Streaming with vector
- No local memory buffer
- Streaming can be stalled from upstream or downstream, It is varied, but approximate 1

```
#include <hls_vector.h>
using vint16 = hls::vector<int, 16>;

void wide_vadd( hls::stream<vint16>& a, hls::stream<vint16>& b,
               hls::stream<vint16>& c )

    for (int i = 0; i < (N/16); i++) {
        vint16 av = a.read();
        vint16 bv = b.read();
        vint16 cv;
        #pragma HLS PIPELINE
            cv = av + bv;
            c.write(cv) ;
    } }
```

# Memory IO Usage

- Transfer large blocks of data from host to global memory
  - Avoid extra copy scatter/Gather due to fragmented host memory, use DPDK (Memory-Pool manager)
- Only copy data back to host when necessary
- Host/Kernel to kernel data transfer, use with priority as below
  1. Stream (dataflow – low latency and low resource usage)
  2. On-chip memory (cache/buffering techniques)
  3. Global memory ( pipeline scheduling to hide transfer latency)
  4. Avoid to use host memory as intermediate data buffer

# Performance Optimization

System Analysis

Verify Design

Synthesize Design

1. Initial Design

2. Optimize Performance

3. Reduce Latency

4. Improve Area

Optimize Host

# Three Optimization Goals

- Enable Pipelining with  $II = 1$  (Tasks parallelism)
  - Ensure all pipelines run at maximum throughput
- Scaling/folding (Data Parallelism)
  - SIMD
- Memory efficiency
  - Compute bound – saturate pipelines with data from memory
  - Memory bound – maximize bandwidth utilization (width/multi-channel/stripping/prefetch)



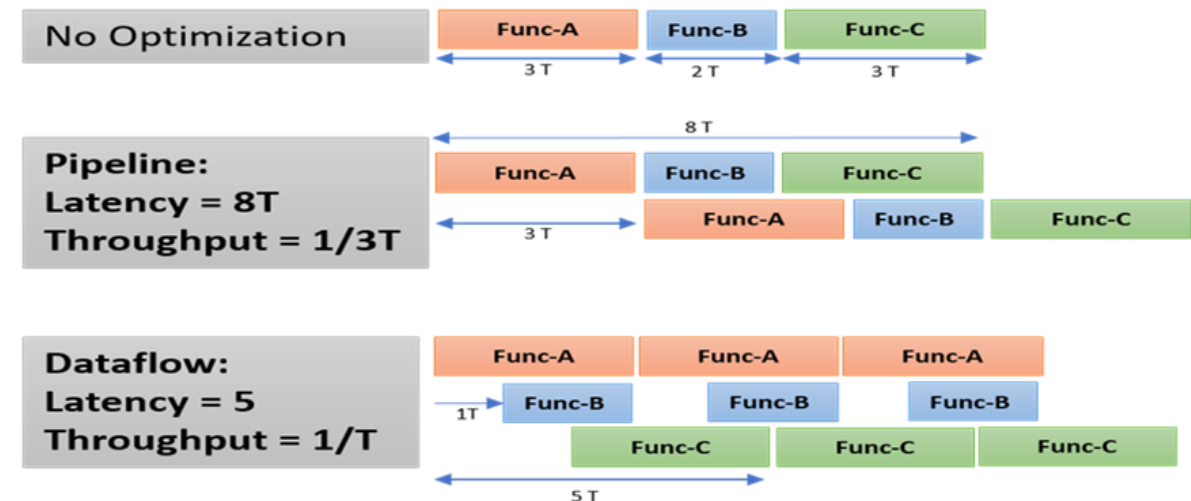
# Basic Concept: Latency, Initiation Interval (II)

- Initiation Interval (II)
  - Number of clock between new input samples
  - Throughput is reciprocal of II
- Iteration Latency (L)
  - # of clock to complete one iteration of a loop
- Loop Latency ( C )
  - # of clock to execute all the iteration in a loop
- N inputs

$$C = L + II \cdot (N-1) \text{ [cycles]}$$

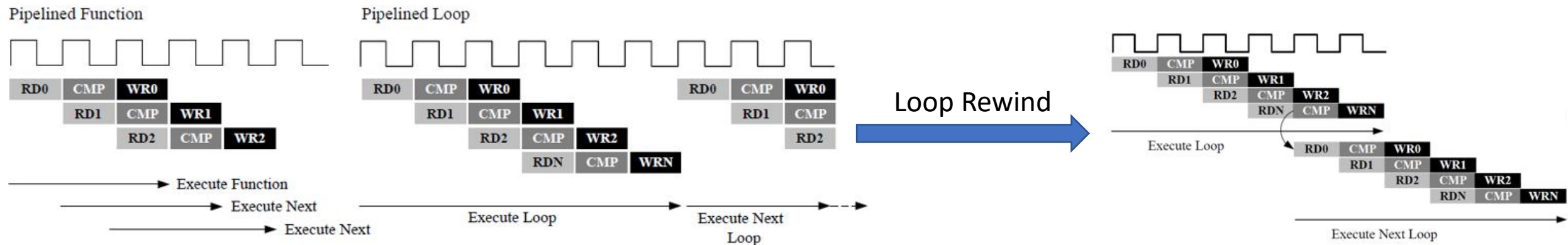
- When  $N \gg L$ , the ultimate goal is to achieve  $II = 1$  (most critical performance metric)

```
Void top(a,b,c,d) {  
    Func_A(a,b,i1);  
    Func_B(c,i1,i2);  
    Func_C(i2,d);  
}
```



# Difference of Function and Loop Pipelining

- Pipeline Function
  - Runs forever and never ends - the functions are automatically rewind.
  - It could unroll all loops in function body **(Avoid pipeline at function level)**
  - Use dataflow parallelism instead
- Pipeline Loop
  - Loop complete all operations in the Loop before starting the next Loop
  - Loop takes overhead (cycles) to enter the Loop, flush the pipeline before entering the next pipeline
  - **Use Loop Rewind**



# Pipeline & Dataflow

- PIPELINE to functions and loops
- Dataflow – functions and loops work in parallel
- Work from the bottom up
  - First, Pipeline sub-function
  - Pipeline a function contains sub-loops
    - Unroll all loops in the hierarchy below (un-favored)
    - Better pipeline the loops in the hierarchy below
- Use Dataflow – pipelined loops operate concurrently
- Pipeline at operator level. (RESOURCE directive for multipliers, adders, block RAM)

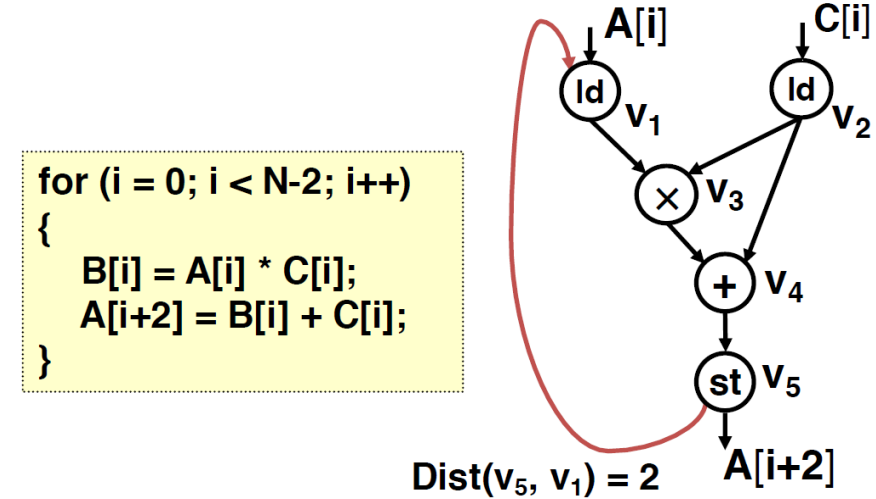
```
function ( ... ) {  
#pragma HLS DATAFLOW  
  
    Pipelined-Loop/function  
  
    ⋮  
  
    Pipeline-Loop/function  
  
}
```

# Issues to Prevent Pipeline, Loop with $II = 1$

- Loops with Variable Bounds
- Data Dependency in inter-iteration – Loop-Carried Dependence
  - RAW (True Dependency), WAR, WAW
- Resource Limitation (Contention)
  - Limited compute resources
  - Limited I/O bandwidth
  - Limited Memory resources, e.g. port limitation

# Calculate II

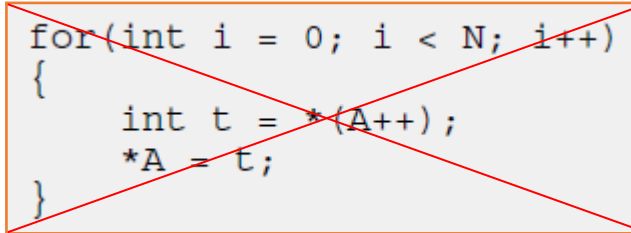
- Dependency Graph
- Latency(c): sum of operation latencies along c
- Distance(c) = iterations separating the two dependent operations
- Ops(r) = # of operation for operator r
- Resource(r) = # of operator available
- ResMII – Minimum II due to Resource Limits
  - $\text{ResMII} = \max [\text{Ops}(r)/\text{Resource}(r) ]$
  - r = operator
- **RecMII – Minimum II due to Recurrences**
  - **$\text{RecMII} = \max [\text{Latency}(c)/\text{Distance}(c)]$**
- $\text{Min-II} = \max(\text{ResMII}, \text{RecMII})$



Latency ->

Iteration->	I/A0	*	*	+	s/A2	
	I/A1	*	*	*	+	s/A3
		I/A2	*	*	*	+

# Compiler limitation on Loop-Carried Dependencies



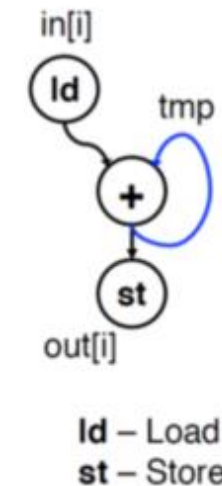
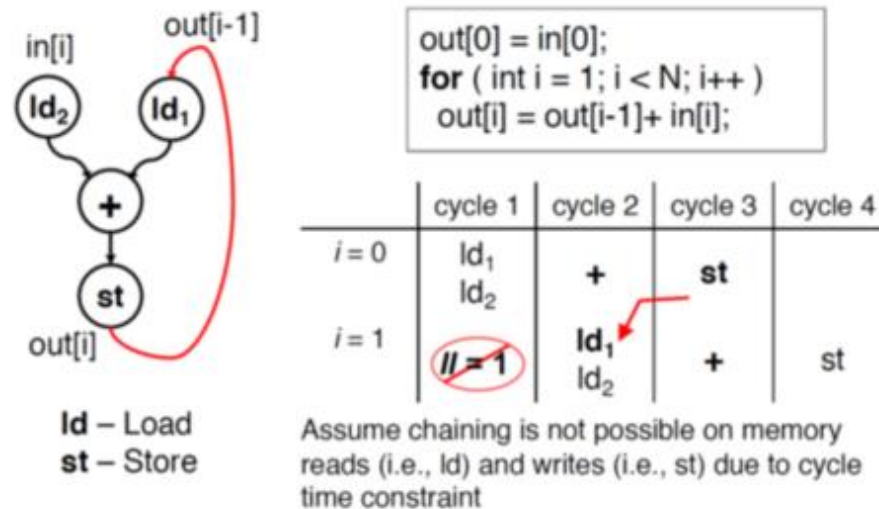
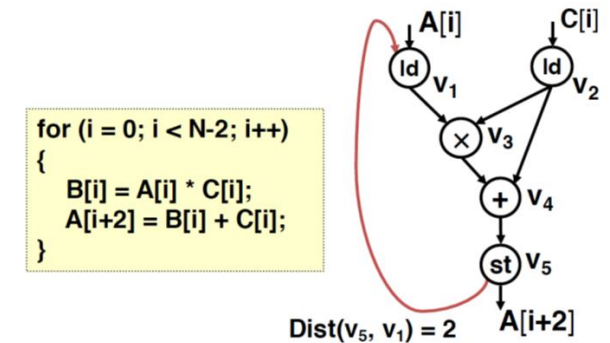
```
for(int i = 0; i < N; i++)
{
    int t = *(A++);
    *A = t;
}
```

- Avoid Pointer Arithmetic
- Use simple array indexes (Constant Bound, Increment by 1)
- Avoid the following
  - Non-constants in array indexes, e.g.  $A[K + i]$  where  $i$  is the loop index and  $K$  is an unknown variable
  - Multiple index variables in the same subscript location, e.g.  $A[i + 2 \times j]$  where  $i$  and  $j$  are loop index variables for a double nested loop
  - Nonlinear indexing, e.g.  $A[i \& C]$ ,  $i$  is a loop variable and  $C$  is a non-constant variable
- Use Loops with Constant Bounds whenever possible
- Ignore Memory Dependence – use HLS Dependence false

# Techniques to improve II

# Data Forwarding – Shorten Latency (c)

- Idea: reduce Latency( c ) in RecMII = max [Latency(c)/Distance(c)]
- Data forwarding
  - A result passes directly to the functional unit that requires it without going through input/output, i.e., memory access.
  - Use temporary registers

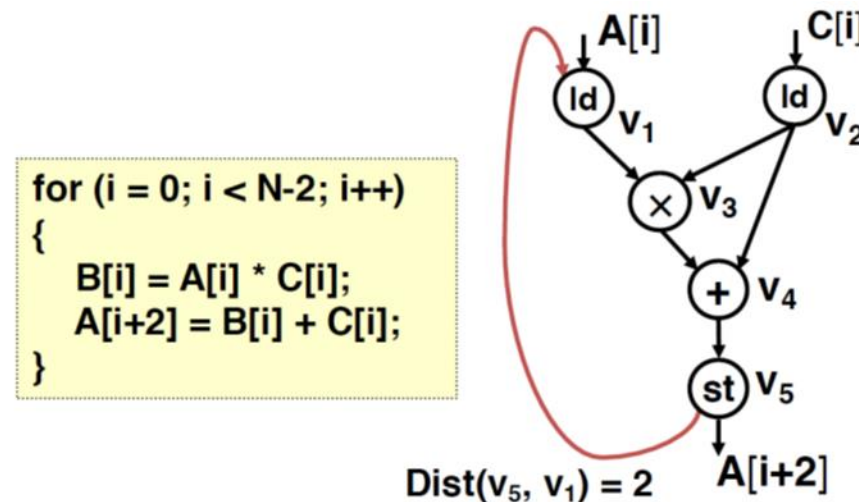


	cycle 1	cycle 2	cycle 3	cycle 4
i = 0	ld	+	st	
i = 1	// = 1	ld	+	st



# Reduce II – Increase the Distance of dependent variables

- $\text{RecMII} = \max [\text{Latency}(c)/\text{Distance}(c)]$
- Apply Tile Interleaving to increase Distance
- The following code



**datatype is double, it takes 6T for \*/+ => II = 6**

```

for(int i = j+1; i < diagSize; ++i){
  dataType tmp2=0;
  Loop vec mul:
  for(int k = 0; k < j; k++){
    tmp2 += dataA[i][k]*dataA[j][k];
  }
  dataA[i][j] = (dataA[i][j] - tmp2)/dataA[j][j];
}
    
```

**Distance = 16, so II = 1**

```

for(int i = j+1; i < diagSize; ++i){
  dataType tmp_i[16] = {0}, tmp3_i, tmp1[8], tmp2[4],
  Loop_vec_mul:
  for(int k = 0; k < j; k++){
    #pragma HLS pipeline
    tmp_i[k % 16] += dataA[i][k]*dataA[j][k];
  }
}
    
```

# Pipeline II=1 for operation takes multiple cycles

- 64-bit double operation for multiple/add takes multiple cycles to complete
- Note: acc is local registers already, data-forward does not help
- Loop transformation - interleave

```
// Matrix-Matrix Multiplication
for (int n = 0; n < N; ++n)
  for (int m = 0; m < M; ++m) {
    double acc = C[n][m];
    for (int k = 0; k < K; ++k) {
      #pragma PIPELINE
      acc += A[n][k] * B[k][m];
    }
    C[n][m] = acc;
  }
```

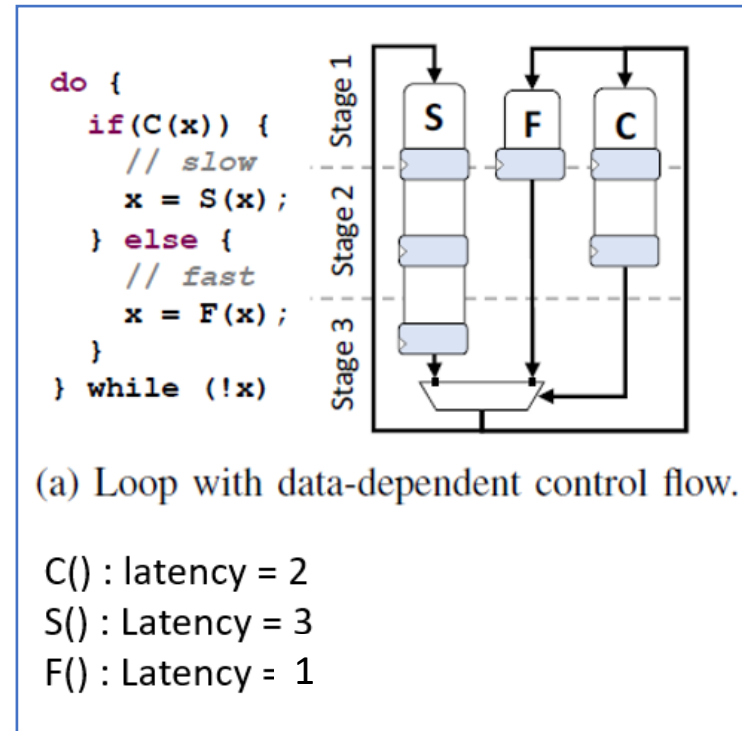
```
for (int n = 0; n < N; n++) {
  for (int m = 0; m < M; m += T) {
    double acc[T]; //Tile of size T
    for (int k = 0; k < K; k++) {
      double a = A[n][k] // M/T read
      for (int t = 0; t < T; t++) {
        #pragma HLS PIPELINE
        double prev = (k == 0) ? C[n][m + t] : acc[t];
        acc[t] = prev + a * B[k][m + t];
      }
    }
    for (int t = 0; t < T; t++) // write out
      C[n][m + t] = acc[t];
  }
}
```

# Speculative Pipeline (detailed in next lecture)

- It can not less than the latency to calculate exit condition
- Loop with data-dependent control flow
- Can we re-structure the code to make  $II = 1$  ?
- Can we still speculative even if there is side effect, e.g. external memory access ?

Exit condition takes multiple cycles

```
while ( m*m*m < N ) {  
    m += 1;  
}
```



# Uncertain and non-uniform dependencies

- Uncertain Dependency

```
for (i=0; i<N; i++)  
    A[i+m] = A[i] + 0.5f;
```

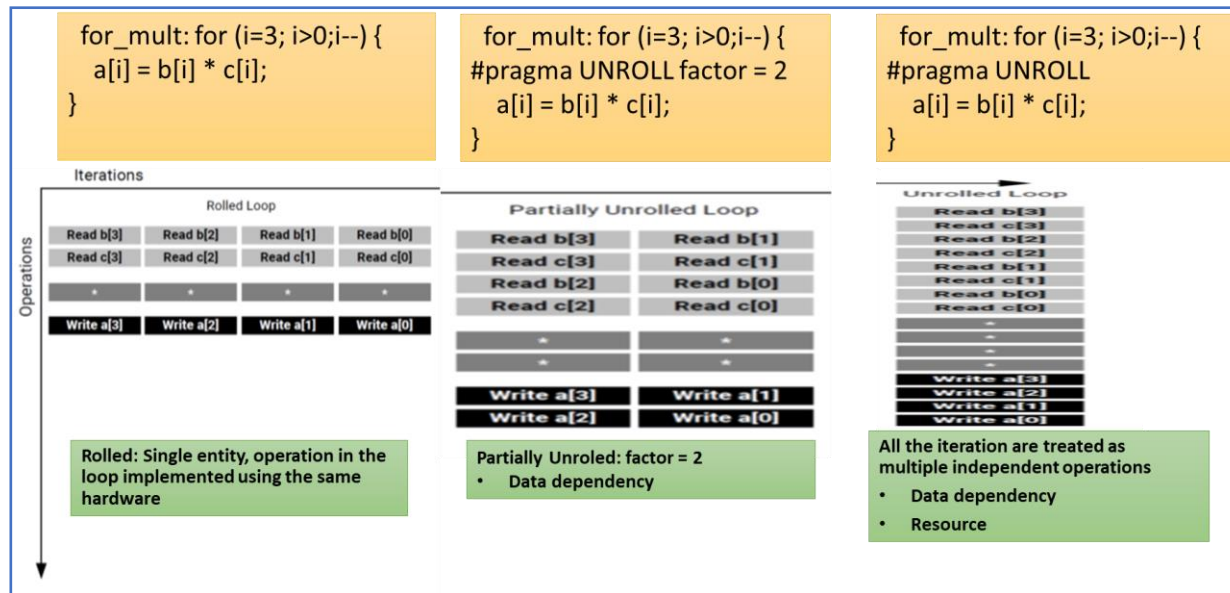
- Value of m is not known at compile time
- Example: Matrix decomposition, triangular matrix computation

- Non-uniform dependency

```
for (i=0; i<N; i++)  
    A[2*i] = A[i] + 0.5f;
```

# Parallelize Loops - Unroll

- Unroll – multiple iterations of a loop execute concurrently
  - Dependencies between loops
  - Resource limitation – memory ports
- Use vector datatype – SIMD
- Increase DRAM bandwidth – use all memory bandwidth available, including
  - memory width 512-bit data-width,
  - multiple memory channels



```
#include <hls_vector.h>
using vint16 = hls::vector<int, 16>;
```

```
void wide_vadd(vint16 *a, vint16 *b, vint16 *c, int size) {
#pragma HLS INTERFACE m_axi bundle=in_a port=a depth=1024
#pragma HLS INTERFACE m_axi bundle=in_b port=b depth=1024
#pragma HLS INTERFACE m_axi bundle=out_c port=c depth=1024
```

```
    for (int i = 0; i < (size/16); i++) {
        c[i] = a[i] + b[i];
    }
}
```

# Function Execute Concurrently v.s. Loop Execute Sequentially.

```
void simple_test(const float arr_in[8], float arr_out[8]) {  
#pragma HLS ARRAY_PARTITION variable=arr_out complete dim=1  
#pragma HLS ARRAY_PARTITION variable=arr_in complete dim=1  
  
    outer: for (int i = 0; i < 8; i += 1) {  
#pragma HLS UNROLL  
        float b = sqrt(arr_in[i]); ;  
        inner: for (int j = 0; j < 4; j++) {  
            b += sqrt(b) ;  
        }  
        arr_out[i] = b ;  
    }  
}
```

```
float f1(const float a)  
{  
    float b = sqrt(a); ;  
    inner: for (int j = 0; j < 4; j++) {  
        b += sqrt(b) ;  
    }  
    return(b);  
}  
  
void simple_test(const float arr_in[8], float arr_out[8]) {  
#pragma HLS ARRAY_PARTITION variable=arr_out complete dim=1  
#pragma HLS ARRAY_PARTITION variable=arr_in complete dim=1  
    outer: for (int i = 0; i < 8; i += 1) {  
#pragma HLS UNROLL  
        arr_out[i]=f1(arr_in[i]) ;  
    }  
}
```

# Executing a Sequence of Multiple Loops in Parallel

- Loop Merge
- Run sequenced loops concurrently by creating a function if no data dependency
- Dataflow on loops

```
void foo() {  
    // first loop  
    for( int i = 0; i < n; i++) {  
        // Do something  
    }  
    // second loop  
    for( int i = 0; i < m; i++) {  
        // Do something else  
    }  
}
```

```
void first_loop() {  
    for( int i = 0; i < n; i++) {  
        // Do something  
    }  
}  
void second_loop() {  
    for( int i = 0; i < m; i++) {  
        // Do something else  
    }  
}  
void foo() {  
    first_loop();  
    second_loop();  
}
```

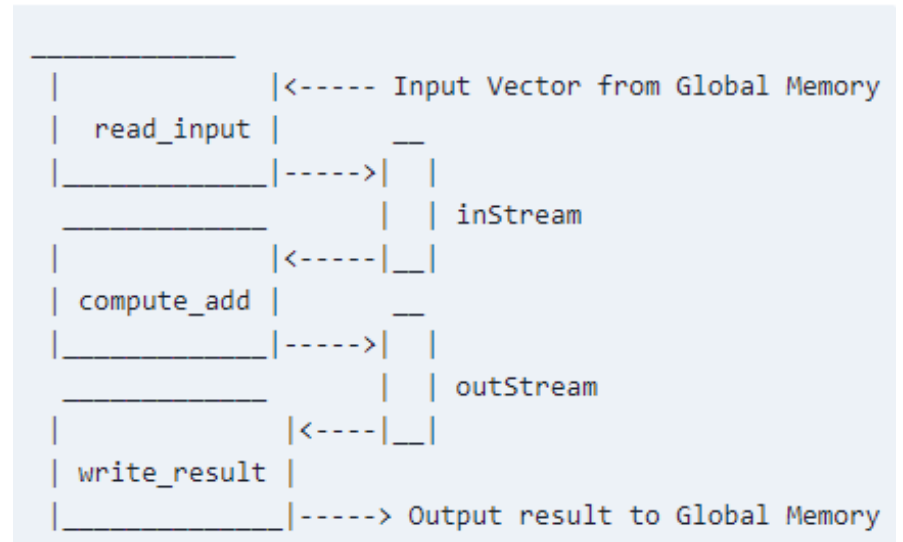
# Dataflow



# Dataflow Canonical Form - Function

- Don't in-line leaf function.
- #pragma HLS STREAM – FIFOs are used
- #pragma HLS DATAFLOW

```
void adder(unsigned int* in, unsigned int* out, int inc, int size) {  
    static hls::stream<unsigned int> inStream("input_stream");  
    static hls::stream<unsigned int> outStream("output_stream");  
    #pragma HLS STREAM variable = inStream depth = 32  
    #pragma HLS STREAM variable = outStream depth = 32  
    #pragma HLS dataflow  
    read_input(in, inStream, size);  
    compute_add(inStream, outStream, inc, size);  
    write_result(out, outStream, size);} }
```



# Dataflow Canonical Form - inside a Loop Body

## Requirements:

- Initial value declared in the loop header and set to 0.
- The loop condition is a positive numerical constant or constant function argument.
- Increment by 1
- Dataflow pragma needs to be inside the loop.
- Should have a single loop counter.
- The inner loop needs to pipeline

```
wr_loop_j: for (int j = 0; j < TILE_PER_ROW; ++j) {
```

```
    #pragma HLS DATAFLOW
```

```
    wr_buf_loop_m: for (int m = 0; m < TILE_HEIGHT; ++m) {
```

```
        wr_buf_loop_n: for (int n = 0; n < TILE_WIDTH; ++n) {
```

```
            #pragma HLS PIPELINE
```

```
            // should burst TILE_WIDTH in WORD beat
```

```
            outFifo >> tile[m][n];
```

```
        } }
```

```
wr_loop_m: for (int m = 0; m < TILE_HEIGHT; ++m) {
```

```
    wr_loop_n: for (int n = 0; n < TILE_WIDTH; ++n) {
```

```
        #pragma HLS PIPELINE
```

```
        outx[TILE_HEIGHT*TILE_PER_ROW*TILE_WIDTH*i
```

```
            +TILE_PER_ROW*TILE_WIDTH*m+TILE_WIDTH*j+n] = tile[m][n];
```

```
    } }
```

```
}
```

**Check if it meets all the requirements**

# Use ap\_ctrl\_none for Dataflow

```
#pragma HLS interface ap_ctrl_none port=return
```

- All processes specified ap\_ctrl\_none
- All processes executed or stalled based on the availability of data in the FIFO, no synchronization on the block level.
- Processes are executed different rate, balance throughput
  - Faster process distributed work to several slower ones

```
void region(...) {  
    #pragma HLS dataflow  
    #pragma HLS interface ap_ctrl_none port=return  
    hls::stream<int> outStream1, outStream2;  
  
    demux(inStream, outStream1, outStream2);  
    worker1(outStream1, ...);  
    worker2(outStream2, ....);  
}
```

```
demux: II = 1  
worker1: II = 2  
worker2: II = 2
```

# Dataflow Summary

- Canonical Form
  - Interface:
    - **Top: #pragma HLS INTERFACE port=xxx axis**
  - Inner functions interface: **static hls::stream<T> variable**
  - Specify depth: **#pragma STREAM variable=xxx depth=x**
- Multiple Levels of Hierarchy
  - **#pragma dataflow**
  - **#pragma HLS INLINE (don't inline leaf function)**
- A leaf level function, apply PIPELINE with enable\_flush
  - **#pragma HLS PIPELINE II=1 enable\_flush**

# Hierarchy & Streaming kernels Illustration

# Rules for Dataflow

## Rules:

- A sequence of function calls forward (no feedback)
- Variables can have only one reading process and one writing process
- Function return type must be void
- No loop carried dependencies

## Limitations

- Single-producer-consumer violations
- Bypassing tasks
- Feedback between tasks
- **Conditional execution tasks**
- Loop with multiple exit conditions

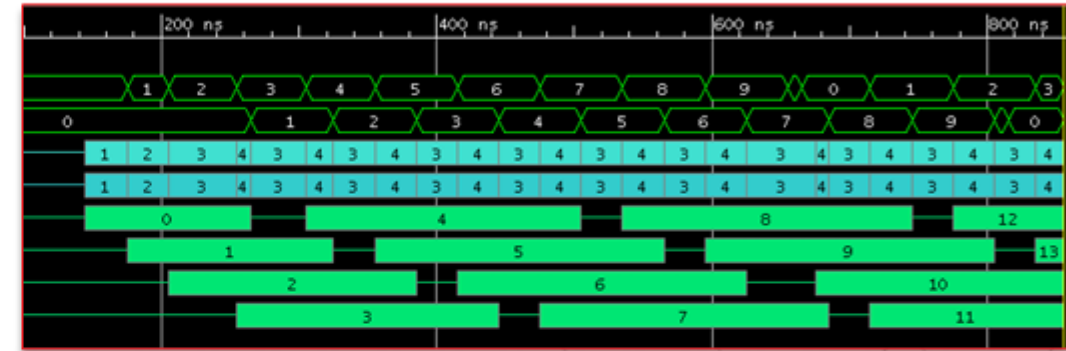
# Dataflow Usage Techniques

- Separate Read-Compute-Write
  - Read/Write – Stream from/to AXI-Memory once (minimize IO read/write)
  - Compute kernels interface with stream (no memory buffer)
- Use small localized cache for
  - Repeated access data
  - Irregular access pattern
- Perform conditional branching
  - Inside pipelined tasks rather than conditionally execute tasks
  - Extensive use of conditional operations in the loop, does not impact pipeline
- Using `hls::stream` to enforce good coding practices

# Analyze Simulation Waveforms

## > New Dataflow waveform viewer(\*)

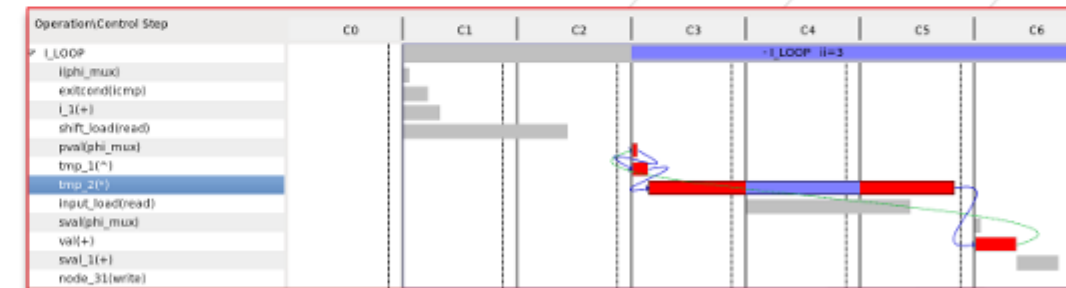
- >> Shows task-level parallelism
- >> Confirm optimizations took place



Co-Simulation Waveforms in v2018.2

## > HLS Schedule Viewer

- >> Shows operator timing and clock margin
- >> Shows data dependencies
- >> X-probing from operations to source code



HLS Schedule Viewer in v2018.2

2018.2: Visible when Dataflow is applied, all traces dumped, using Vivado simulator and checking waveform debug



# Latency Optimization

System Analysis

Verify Design

Synthesize Design

1. Initial Design

2. Optimize Performance

3. Reduce Latency

4. Improve Area

Optimize Host

# Using pragma latency

- Set Min/Max latency functions, loops, and regions  
`#pragma HLS latency min=<int> max=<int>`

Function latency

```
int foo( ... ) {  
#pragma HLS latency min=4 max=8  
    char y;  
    y = x*a+b+c;  
    return y  
}
```

Loop\_1 – max latency 12

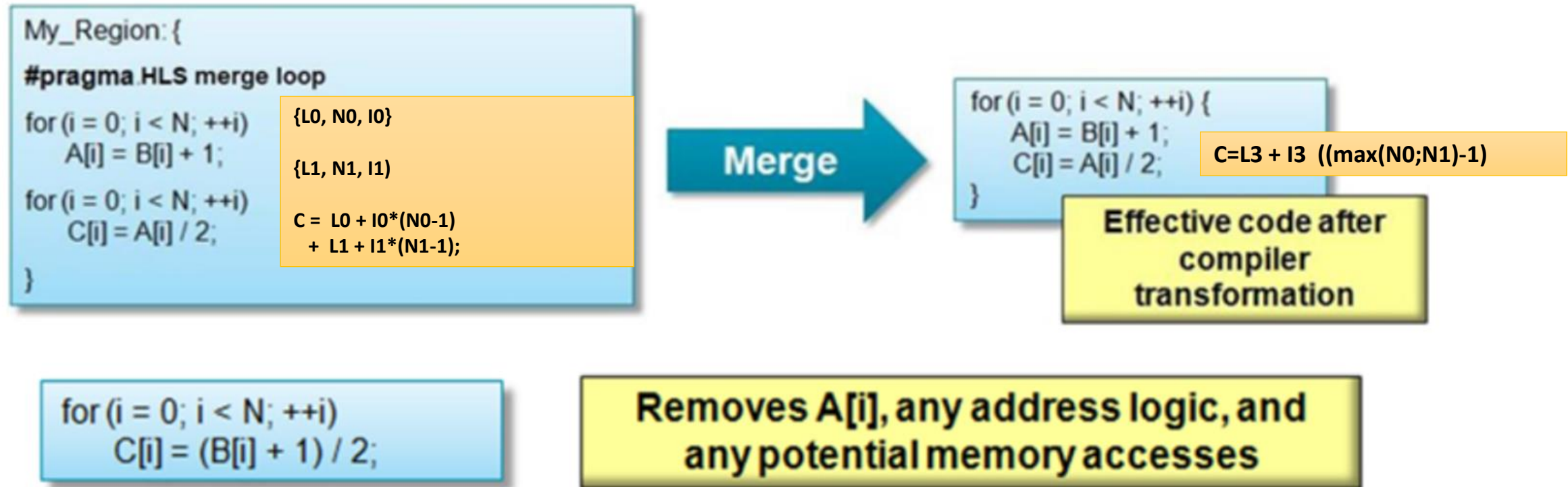
```
loop_1: for(i=0;i< num_samples;i++) {  
#pragma HLS latency max=12  
    ...  
}
```

Group signals that need to change in the same clock cycle by latency = 0

```
// create a region { } with a latency = 0  
{  
#pragma HLS LATENCY max=0 min=0  
*data = 0xFF;  
*data_vld = 1;  
}
```

# Merge Sequential Loop

- Merge sequences of Loops -> loops run in parallel
- Reduce # of clocks transition between loop-body
- Allow logic within loops to be optimized together and implemented parallel



# Loop Merge Rules

1. If loop bounds are all variables, they must have the same value
2. If loop bounds are constant, the maximum constant value is used as the bound of the merged loop
3. Loops with both variable bound and constant bound cannot be merged
4. Code between loops to be merged cannot have side effects: multiple execution of this code should generate the same results (  $a=b$  is allowed;  $a=a+1$  is not)
5. Loops cannot be merged when they contain FIFO accesses. Merging would change the order of the reads and writes from a FIFO

# Flattening Nested Loops

- #pragma HLS loop\_flatten [off]
- **It costs a clock cycle to move between loops in the loop hierarchy**
- Reducing # of cycles for loop operation
- Flattens from the loop with #pragma HLS loop\_flatten and all (perfect or semi-perfect) loops above into a single loop
- **Apply LOOP\_FLATTEN to the loop body of the inner-most loop**

```
Loop_I: for(i=0;i<20;i++) {  
  Loop_J: for(j=0;j<20;j++) {  
    #pragma HLS loop_flatten  
    operation(i,j);  
  }  
}
```

After flatten →

```
Loop_I: for(k=0;k<400;k++) {  
  i = k / 20;  
  j = k % 20;  
  operation(i, j);  
}
```

# Construct Well-Formed Loops

- Well-formed loop, e.g. `for(int i = 0; i < N; i++ )`
  - Constant bound
  - Increment by 1
- Perfect loop

```
Loop_outer: for (i=3;i>=0;i--) {  
    Loop_inner: for (j=3;j>=0;j--) {  
        [Loop body]  
    }  
}
```



## Perfect Loop

- Only innermost loop has loop body
- No logic between the loop
- All loop bounds constant

```
Loop_outer: for (i=3;i>N;i--) {  
    Loop_inner: for (j=3;j>=0;j--) {  
        [Loop body]  
    }  
}
```

## Semi-perfect Loop

- Outermost loop bound a variable

```
Loop_outer: for (i=3;i>N;i--) {  
    [Loop body]   
    Loop_inner: for (j=3;j>=M;j--) {  
        [Loop body]   
    }  
}
```

## Imperfect Loop

- Loop body is not exclusively inside the innermost loop
- The inner loops has variable bounds

Can be flattened

# Place if-statement in the lowest possible Scope in a nested loop

- Avoid placing loops within conditional statements
- Condition loop prevent effective pipelining the loops & rewind
- Note: Contradict to coding for CPU which branch creates pipeline stall

```
for (int row = 0; row < outerTripCount; row++) {  
    if (loopCondition) {  
        for (int col = 0; col < innerTripCount; col++) {  
            foo(); } }  
    } else {  
        for (int col = 0; col < innerTripCount; col++) {  
            bar(); }  
    }  
}
```



```
for (int row = 0; row < outerTripCount; row++) {  
    for (int col = 0; col < innerTripCount; col++) {  
        if (loopCondition) {  
            foo();  
        } else {  
            bar();  
        }  
    }  
}
```

# Simplify Loop-Exit Conditions

- Avoid complex loop-exit conditions
  - Subsequent iterations of the loop cannot launch in the loop pipeline until the evaluation completes
- Avoid multiple loop-exit

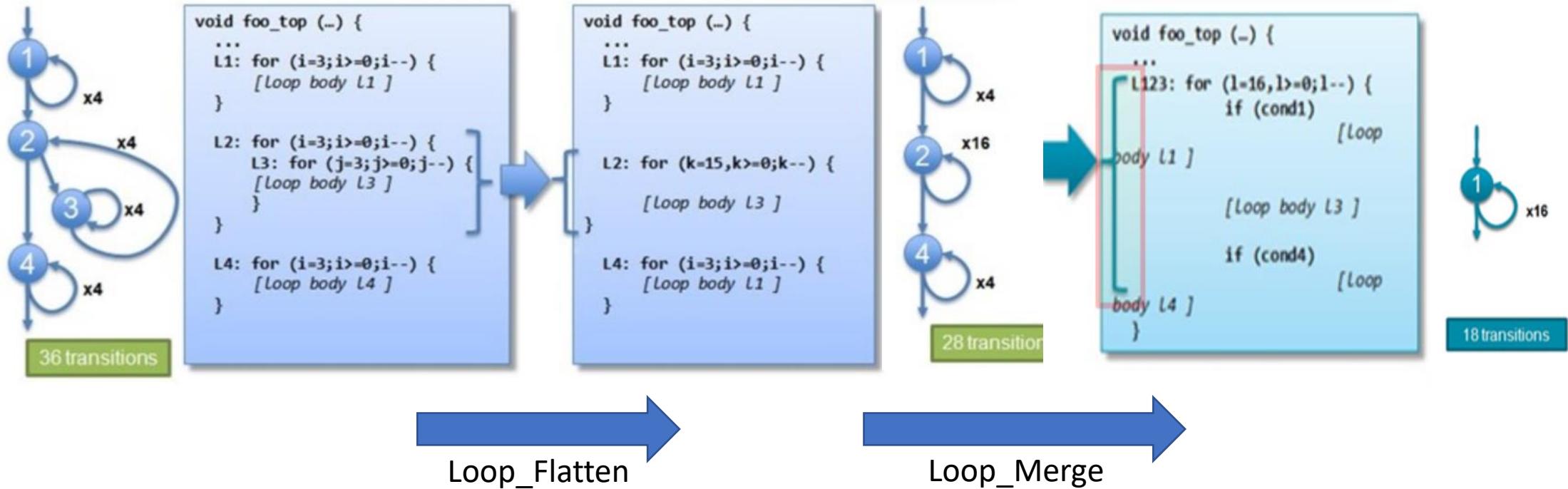
```
Top (...)  
decoderInput_whileloop:while(1)  
{  
  #pragma HLS PIPELINE II=1 rewind  
  if(!inputStream.empty()) {  
    if(cnt==0) {  
      } else {  
        return 0;    // multiple exit condition  
      }  
    } else {  
      return 1;    // multiple exit  
    }  
  }  
  ...  
}
```



```
return_value = 1;  
if(condition) {  
  ...  
  return_value = 0;  
}  
return return_value;
```



# Loop Latency Optimization - Flatten & Merge



# Area Optimization

System Analysis

Verify Design

Synthesize Design

1. Initial Design

2. Optimize Performance

3. Reduce Latency

4. Improve Area

Optimize Host

# Area Optimization

- Allocation
- Bind-op, Bind-storage
- INLINE
- LOOP\_MERGE
- ARRY\_MAP
- Arbitrary Precision Type
- Logic Optimization
- Multi-pumping

# Directive ALLOCATION

- Limiting the number of functions, operators, core
- #pragma HLS ALLOCATION**  
    **instances=<list> limit=<value> <type>**  
    <type>: function, operation
- Globally minimizing Operators - Use config\_bind

```
void foo();  
void foo_top(  
    #pragma HLS ALLOCATION instances=foo limit=1 function  
    foo()  
    foo()  
    foo()  
    foo()  
)  
{
```



```
dout_t array_arith(dio_t d[317]) {  
    static int acc; int l;  
    #pragma HLS ALLOCATION instances=mul limit=16 operation  
    for (i=0; i<317;i++) {  
        #pragma HLS UNROLL  
        acc += acc *d[i];  
    }  
    return acc;  
}
```

# BIND\_OP

- Control the specific resource (core) used to implement the operation
- **#pragma HLS bind\_op variable=<variable> op=<type> impl=<value> latency=<int>**  
op=<type> : mul, add, sub  
Impl=<value>: fabric, and dsp

```
#pragma HLS BIND_OP variable=c op=mul impl=fabric latency=2  
c = a * b; // multiplication at 2T  
d = a * c; // let HLS to decide
```

```
#pragma HLS BIND_OP variable=temp op=add impl=dsp  
temp = inB + inA;  
*out1 = temp;
```

- Multiple operations on a single line. Create a temp variable to isolate the specific operation, e.g.  $a = b + c * d$ ;

```
#pragma HLS RESOURCE variable=temp latency=2  
temp = c * d;  
a = b + temp
```

# BIND\_STORAGE

- assigns a variable (array, or function argument) in the code to a specific memory type (type) in the RTL. e.g. a single or a dual-port RAM
- **#pragma HLS bind\_storage variable=<variable> type=<type> [ impl=<value> latency=<int> ]**

type=<type>: include: fifo, ram\_1p, ram\_1wnr, ram\_2p, ram\_s2p, ram\_t2p, rom\_1p, rom\_2p, rom\_np.

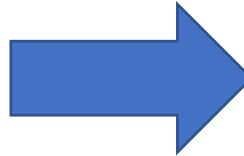
impl=<value>: bram, bram\_ecc, lutram, uram, uram\_ecc, and srl

```
Void foo_top ( int coeffs[128] )
```

```
#pragma HLS RESOURCE variable=coeffs core=RAM_1P
```

# Reuse Hardware By Calling function In a Loop

```
int foo(int a) { return 4 + sqrt(a); }  
  
void myComponent() {  
    ...  
    int x =  
    x += foo(0);  
    x += foo(1);  
    x += foo(2);  
    ...  
}
```



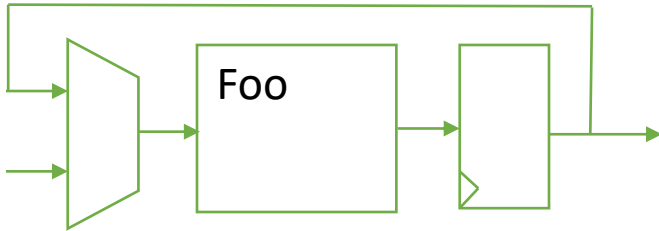
```
void myComponent()  
{  
    ...  
    int x = 0;  
    for (int i = 0; i < 3; i++) {  
        #pragma unroll 1  
        x += foo(i);  
    }  
    ...  
}
```

# Directive INLINE

- Remove function hierarchy
- Better logic optimization, improve the area, better sharing

```
void foo() { .... }  
void foo_top () {  
    foo(...);  
    foo(...);  
}
```

Set\_directive\_allocation -limit 1 -  
type function foo\_top foo



```
void dummy1 (...) { foo( ... ); }  
void dummy2 (...) { foo( ... ); }  
void foo_top(...) {  
    dummy1( ... );  
    dummy2( ... );  
}
```

Set\_directive\_allocation -limit 1 -  
type function foo\_top foo

foo\_top

dummy1

foo

dummy2

foo



set\_directive\_allocate -limit 1 -type  
function foo\_top foo  
set\_directive\_inline dummy1  
set\_directive\_inline dummy2





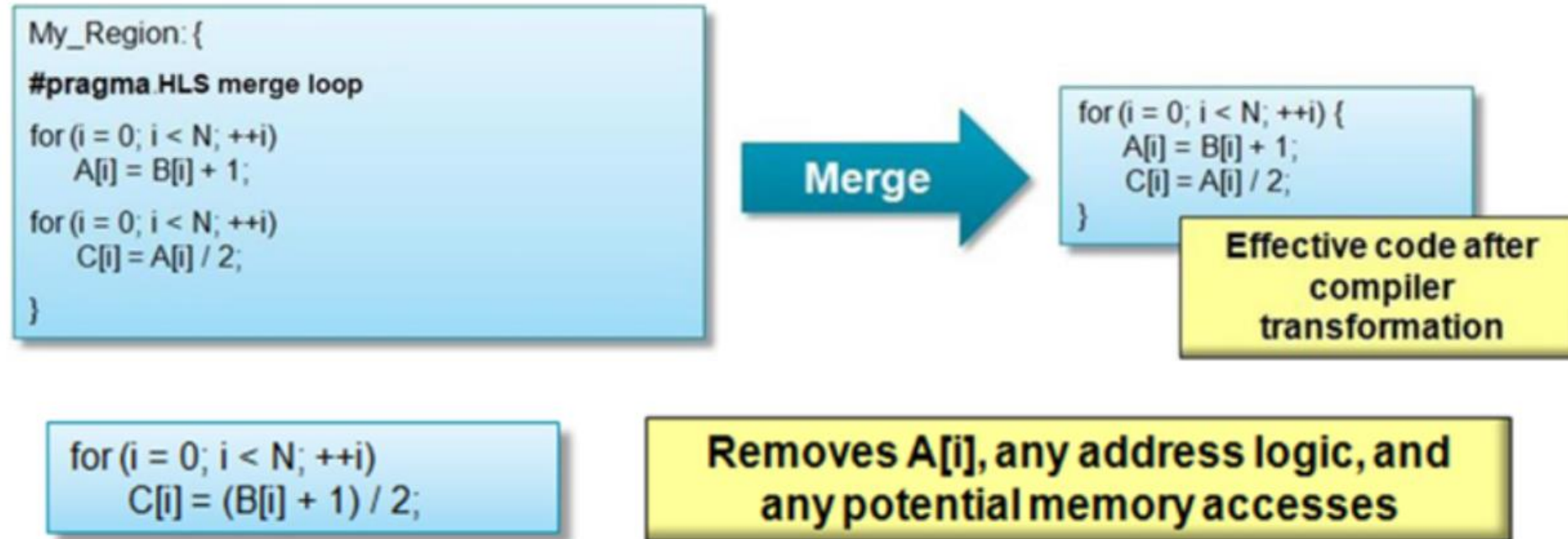
# FUNCTION\_INSTANTIATE

- Create a unique RTL implementation for each instance of a function
- Locally optimized according to function call
- Exploits constant input, simply control structure produce smaller more optimized function blocks

```
char foo_sub(char inval, char incr) {  
    #pragma HLS function_instantiate variable=incr  
    return inval + incr;  
}  
void foo(char inval1, char inval2, char inval3,  
char *outval1, char *outval2, char * outval3) {  
    *outval1 = foo_sub(inval1, 1);  
    *outval2 = foo_sub(inval2, 2);  
    *outval3 = foo_sub(inval3, 3);  
}
```

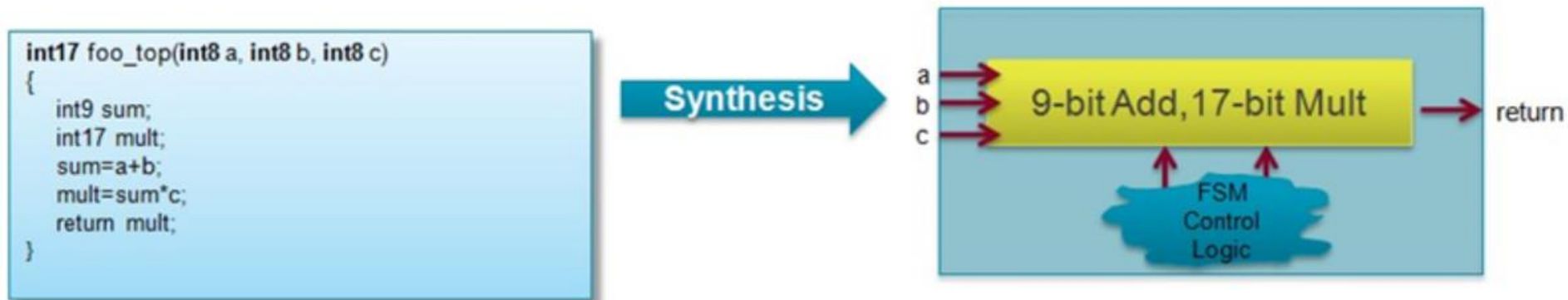
# Loop Merge

- Merging loops allows the logic in the loop to be optimized together
- It can remove the redundant computation among multiple loops



# Arbitrary Precision

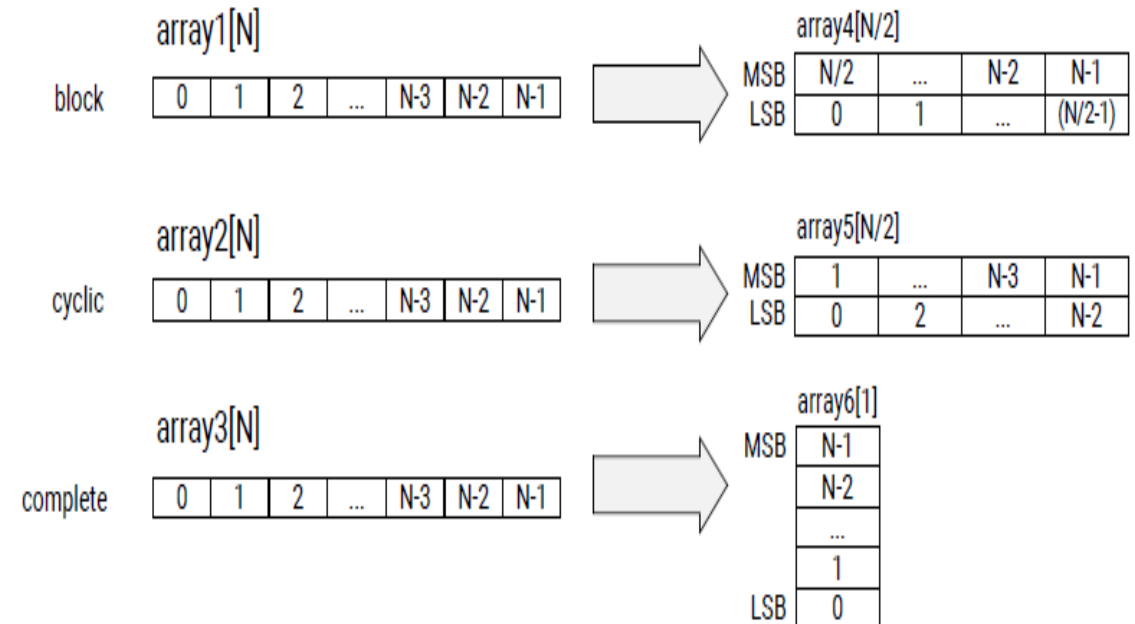
- C/C++ have standard types on the 8-bit boundary
  - char (8-bit), integer(32-bit), long (64-bit)
- HLS supports arbitrary precision bit-width
  - Include “ap\_cint.h”
  - Int<N> signed, uint<N> unsigned
- The full precision can be simulated/validated with C-simulation



# ARRAY\_RESHAPE

- When using UNROLL to increase parallelism, use ARRAY\_RESHAPE allows more data accessed in a single cycle. re

```
void foo (...) {  
  int array1[N];  
  int array2[N];  
  int array3[N];  
  #pragma HLS ARRAY_RESHAPE variable=array1 block  
  factor=2 dim=1  
  #pragma HLS ARRAY_RESHAPE variable=array2 cycle  
  factor=2 dim=1  
  #pragma HLS ARRAY_RESHAPE variable=array3 complete  
  dim=1  
  ...  
}
```



# Logic Optimization

- Operator pipelining
  - BIND\_OP, BIND\_STORAGE –latency
  - Config\_op – pipeline all instance of a specific operation
- Expression balancing – balanced tree , reduce latency
  - For integer – balancing is default on
  - For floating-point – expression balancing is off by default
  - EXPRESSION\_BALNCE off – to save area
- Note: float/double needs maintain the order of operations to match C/C++ simulation result due to saturation and rounding handling

```
data_t foo_top (data_t a, data_t b, data_t c, data_t d) {  
    data_t sum = 0;  
    sum += a;  
    sum += b;  
    sum += c;  
    sum += d;  
    return sum;  
}
```

Figure 72: **Adder Tree**

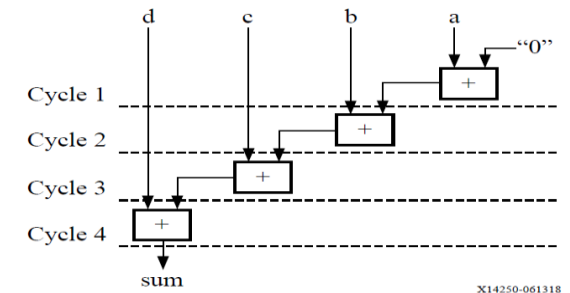
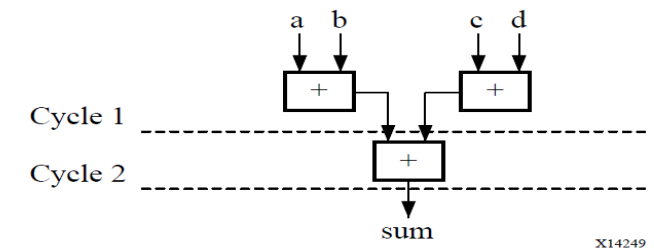


Figure 73: **Adder Tree After Balancing**



# Dataflow: Reduce Channel Resource

- Dataflow memory channels – reduce # of block RAM
- If not streaming data, using ping-pong buffers
  - 2 blockRAM size N: size of the array passed between tasks
- If streaming (using STREAM directive) – FIFO
  - Small depth using registers
  - Producer, consumer same rate  $\rightarrow$  depth = 1
  - Reduce data rate ( $X \rightarrow 1$ ), input depth of X
  - Increase data rate ( $1 \rightarrow Y$ ), output depth of Y
  - If depth is set too small – stall (hang) during hardware emulation

# Declare Variables in the Deepest Scope Possible

- To reduce resource, declare the variable just before you use it in a loop
- Minimize data dependencies and FPGA hardware usage
  - Because it does not need to preserve the variable data across loops that do not use the variables.

# Multi- Pumping

- Hardmacro blocks are typically capable of running at higher frequencies than most designs implemented in the logic fabric
- Multipumping on
  - DSP
  - Multi-port BRAM

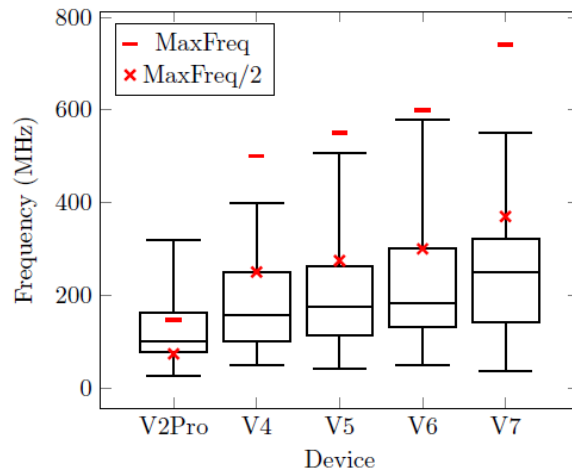


Fig. 1: Reported frequencies on Xilinx Virtex devices for over 350 papers (1100 designs) published in recent FPGA conferences.

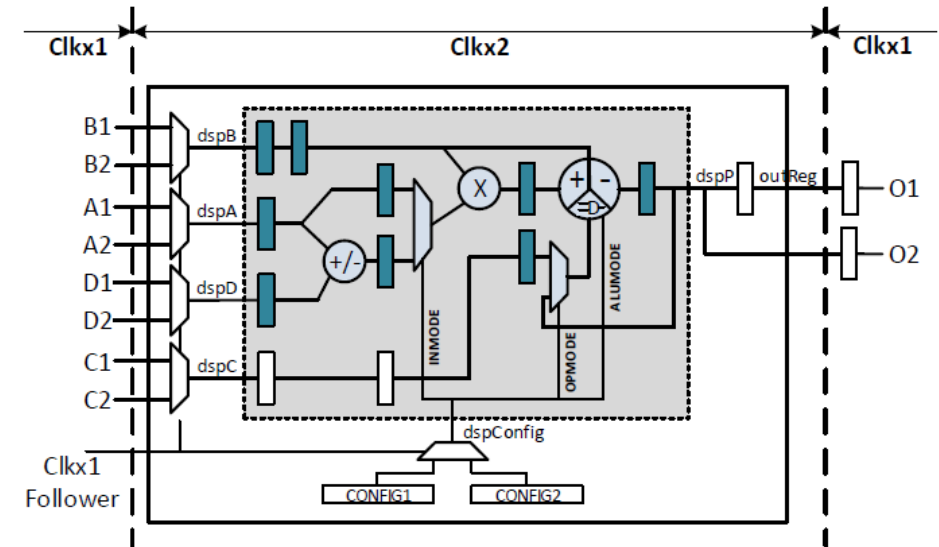
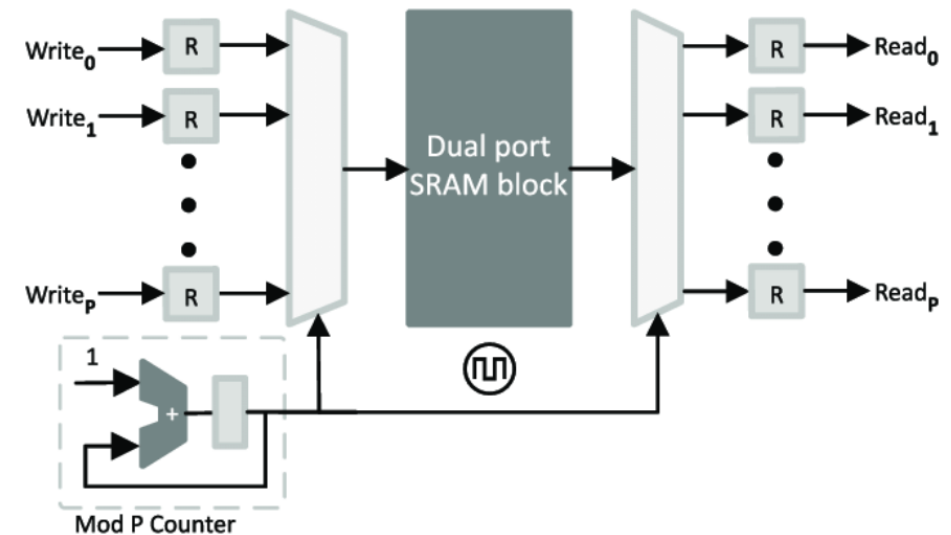


Fig. 4: Multipumped DSP block (mpDSP) architecture.





# Host Optimization

System Analysis

Verify Design

Synthesize Design

1. Initial Design

2. Optimize Performance

3. Reduce Latency

4. Improve Area

Optimize Host

# Host Optimization

- Reduce Overhead of Kernel Enqueueing
- Optimize Data Movement
- Software Parallelism and Kernel Scheduling, how to
  - Reduce host overhead
  - Reduce latency & maximize throughput

# Reduce Overhead of Kernel Enqueueing

- Dispatching command/arguments to kernel takes 30us – 60us (non-posted)
- Minimize call to clEnqueueTask
  - Finish all the workload in a single call
  - Aggregate large data packets
- Auto-Restarting Kernel (Free-Running)
  - Kernel starts determined by availability of data
  - Available in Vitis 2021.2

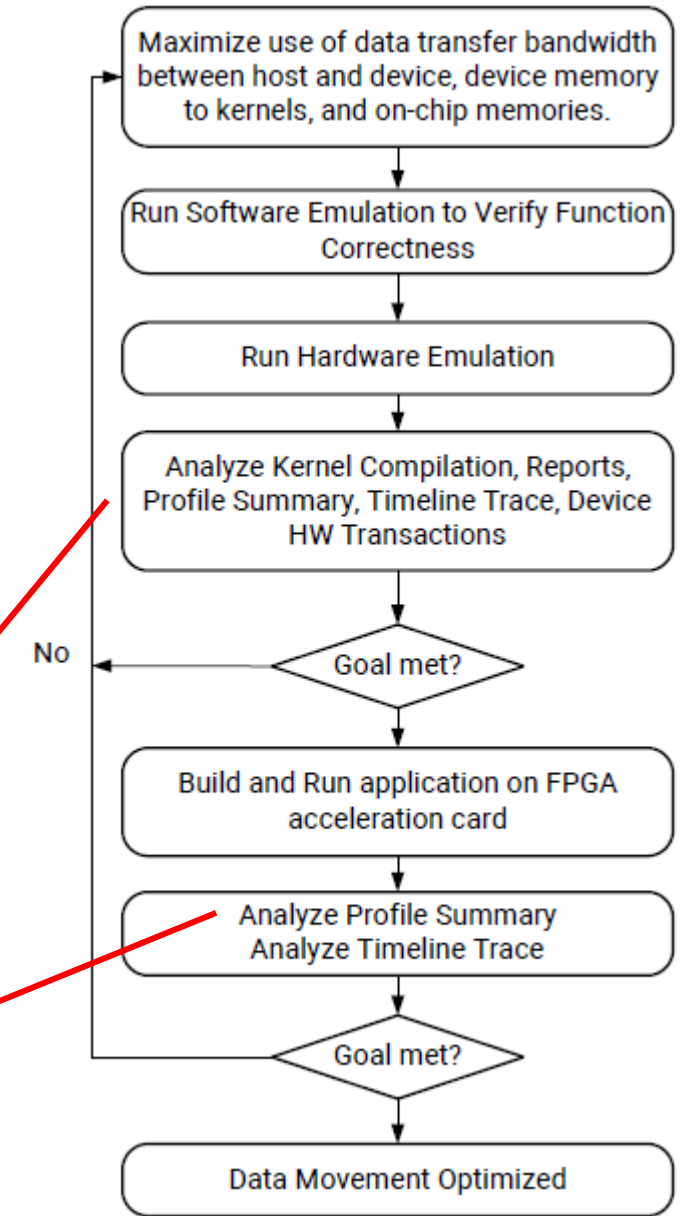
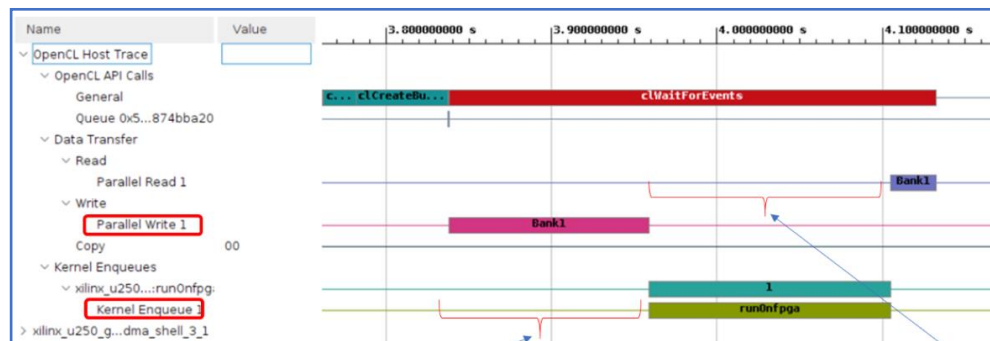
```
void add(int *a , int *b, int inc){  
    int buff_a[SIZE];  
    for(int i=0;i<size;i++) buff_a[i] = a[i];  
    for(int i=0;i<size;i++) b[i] = a[i]+inc;  
}
```

```
void add(int *a , int *b, int inc, int num_batches) {  
    int buff_a[SIZE];  
    for(int j=0;j<num_batches;j++) {  
        for(int i=0;i<size;i++) buff_a[i] = a[i];  
        for(int i=0;i<size;i++) b[i] = a[i]+inc;  
    }  
}
```

# Optimize Data Movement

# Optimizing Data Movement I

- OpenCL computation model:
  - Host Memory -> Global Memory -> (On-chip memory) -> Kernel
  - Maximum all levels of memory interface
- Optimize data movement first before optimizing computation
  - Separate data movement scheduling from computation kernel scheduling
  - Resource availability, dependency resolution



# Models of Data Movements

1. Vitis/OpenCL – Global Memory
2. Kernel Directly Access Host memory
3. Host to Kernel Streaming
  1. Control channel
  2. Data channel
  3. Combined
4. PCIe Peer-to-peer
5. Memory-to-Memory

# Kernel Directly Access Host Memory

- When CPU processing can not keep up with Kernel execution
  - Kernel directly access host memory, CPU does not perform data transfer. Free up CPU cycles
  - Free up CPU helped in processing more Kernel execution requests
- Directly place data in on-chip BRAM – save extra DDR -> BRAM transfer
- Implementation
  - Configure kernel port connect to HOST  
[connectivity] sp=my\_kernel\_1.m\_axi\_gmem:HOST[0] # DDR[0] - Global Memory
  - Enable Host Memory by XRT : xbutil config –host-mem
  - OpenCL – a new buffer extension Flag: XCL\_MEM\_EXT\_HOST\_ONLY

```
cl_mem_ext_ptr_t host_buffer_ext;  
host_buffer_ext.flags = XCL_MEM_EXT_HOST_ONLY;  
host_buffer_ext.obj = NULL;  
host_buffer_ext.param = 0;  
  
cl::Buffer buffer_in (context,CL_MEM_READ_ONLY |CL_MEM_EXT_PTR_XILINX, size, &host_buffer_ext);  
cl::Buffer buffer_out(context,CL_MEM_WRITE_ONLY |CL_MEM_EXT_PTR_XILINX, size, &host_buffer_ext);
```

<https://xilinx.github.io/XRT/master/html/hm.html>

# PCIe Peer-to-Peer

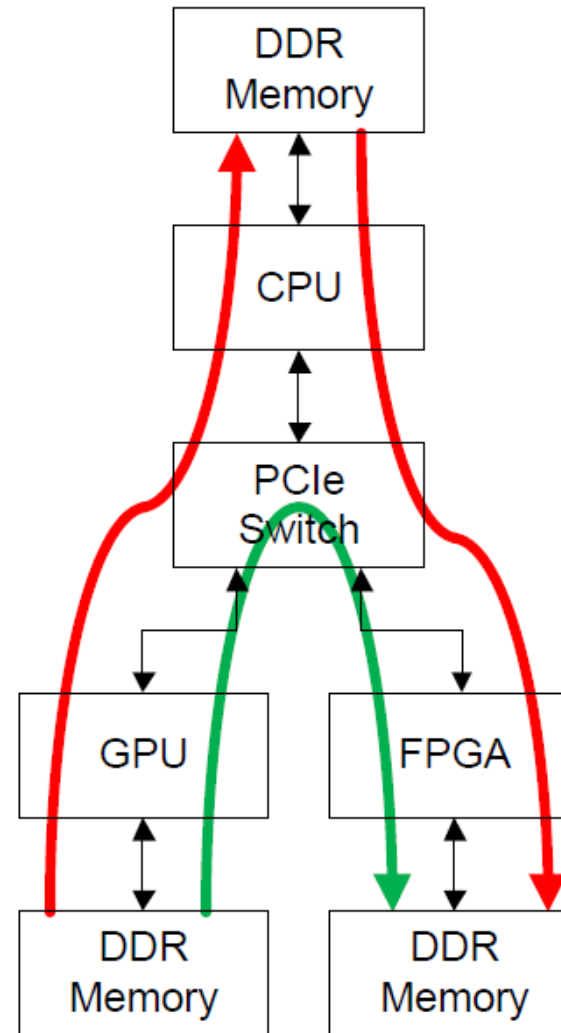


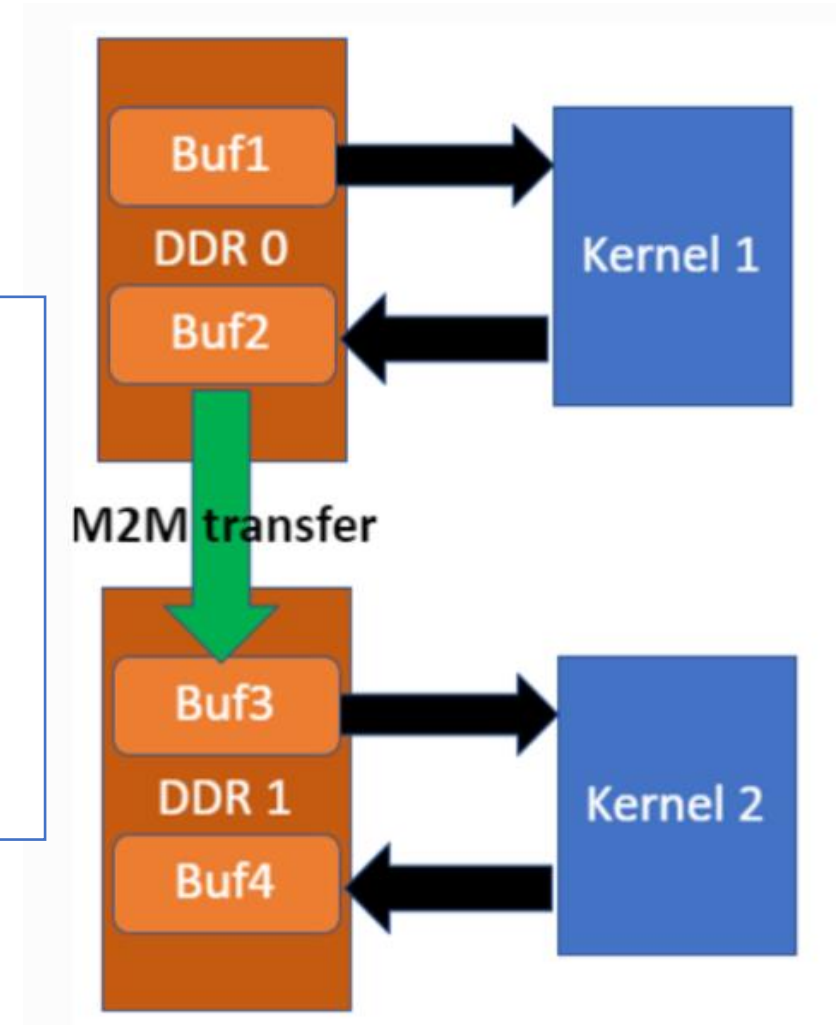
Figure 1. Two Conceptual Models Of GPU To FPGA Transfers



# Memory-to-Memory

- Kernel 1 & Kernel 2 are in different SLR
- Kernel 1 uses DDR0 bank
- Kernel 2 uses DDR1 bank
- The output of the K1 is consumed by K2 for its input.
- So there has to be data transfer from DDR bank0 to DDR bank1.

```
clEnqueueTask(queue, K1 ,0,nullptr,&events1);  
clEnqueueCopyBuffer(queue, Buf2,Buf3,0,0,Buffer_size, 1, &event1, &event2);  
clEnqueueTask(queue,K2,1,&event2,nullptr);
```



# Buffer Memory Fragmentation

- Buffer allocation/deallocation causes fragmentation.
- Kernel execution stalled waiting for continuous space to be freed (OS performs garbage collection)
- Buffer allocated is better continuous in hardware ( or use scatter-gather DMA)
- Allocating device buffer and reusing it. (Intel DPDK uses this technique)

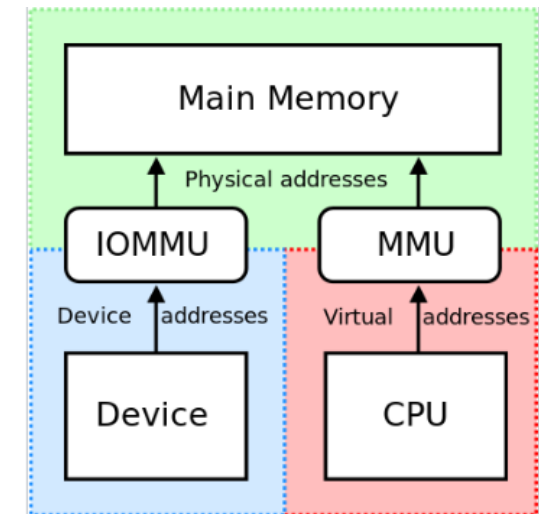
# Access with Fragmented Memory

1. Extra memory copy to continuous buffer
2. Multiple DMA operations (XDMA)
3. Scatter/Gather DMA (QDMA)
4. Contiguous Buffer DMA (Intel DPDK)
5. IOMMU (Device use Virtual Address)

Table 1. Comparison between OS and DPDK allocators.

	Regular Linux* allocator <sup>1</sup>	DPDK rte_malloc
Huge page memory support	Not enforced	Default
NUMA node pinning	Not enforced	Default
Access to IOVA addresses	No	Yes
IOVA-contiguous memory	No	Yes
Cache-aligned allocations	Not enforced	Enforced
Arbitrary alignment for allocations	Yes	Yes
Full shared memory for multiprocessing	No	Yes
Multiprocess thread safety	No	Yes

1. There are third-party libraries that can provide each of these features, but there is not one library that provides all of them.

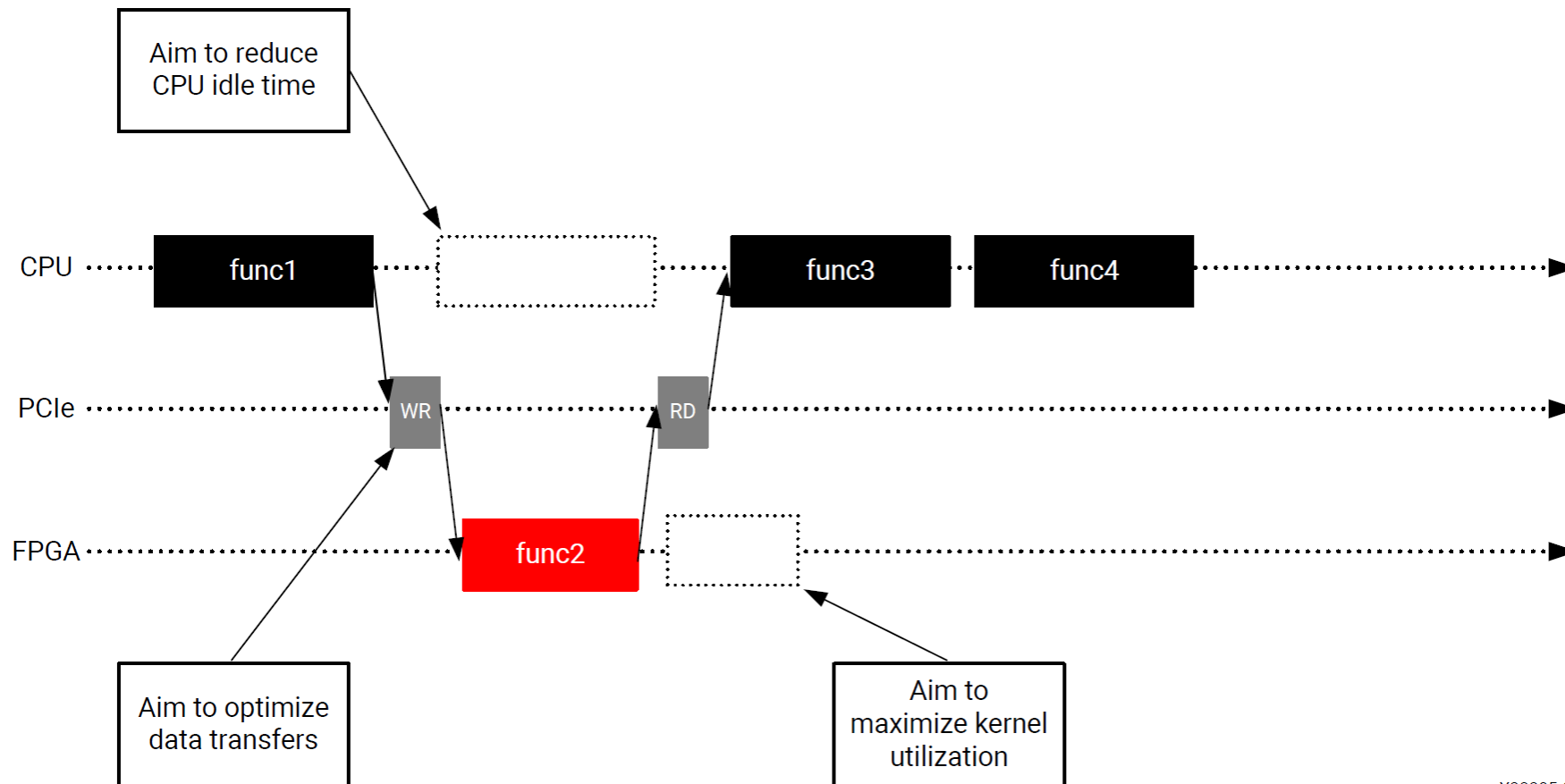


**AMD – Vi, Intel VT-d**

# Access with Fragmented Memory Illustrated

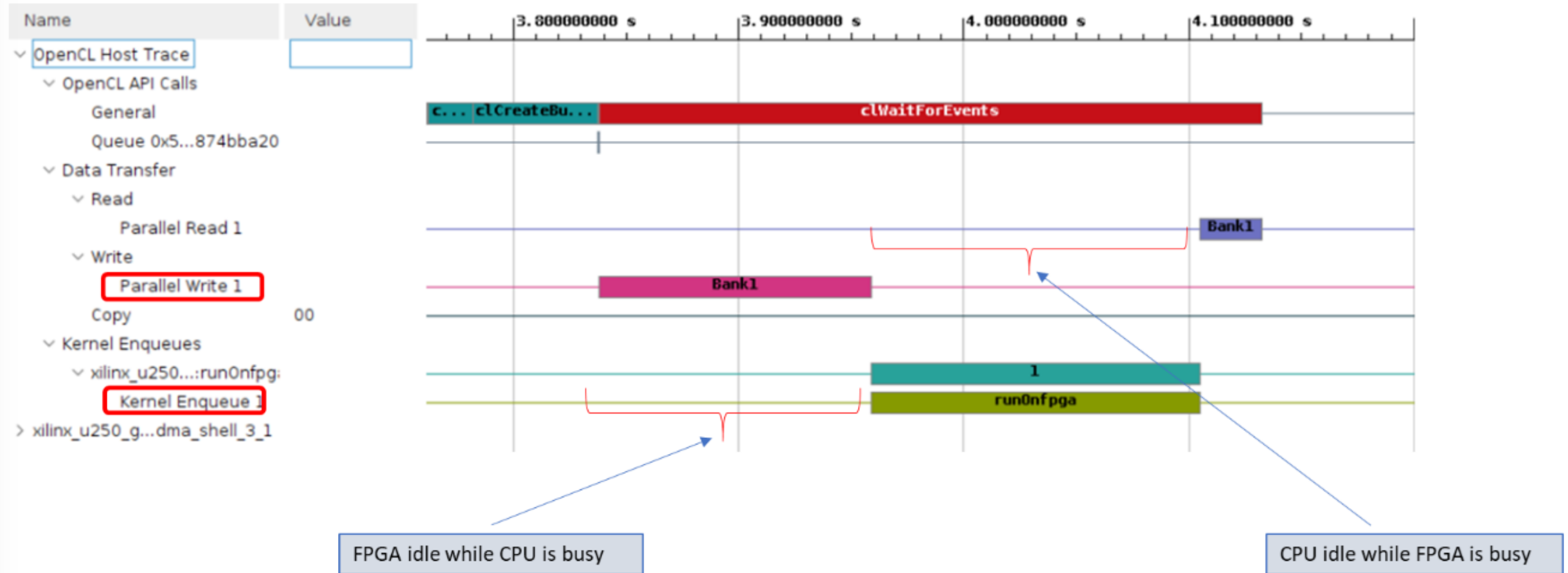
# Software Parallelism & Kernel Scheduling

# Software Application Parallelism



X23285-092619

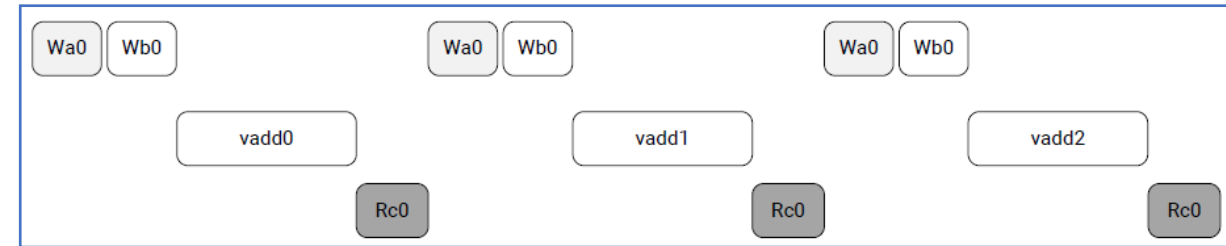
# Figure: Host Application Timeline



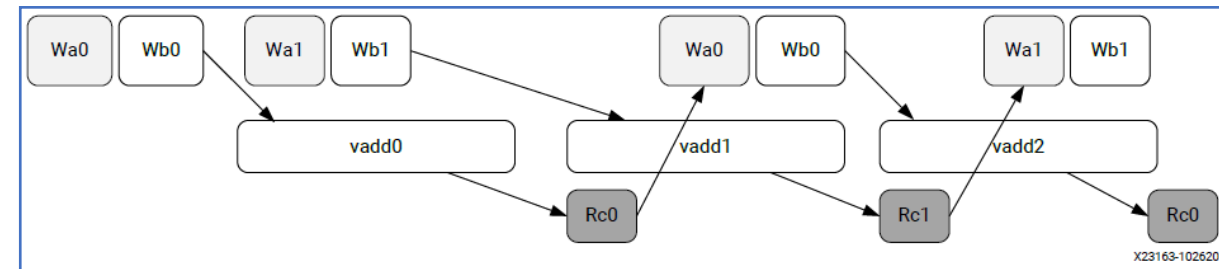
# Overlapping Data Transfers with Kernel Computation

1. Partition big trunk data into small blocks
2. Use out-of-order command queue, with event objects
3. Double buffering
4. Balance IO / Compute throughput
5. Refer to Lab#3

In-order command queue



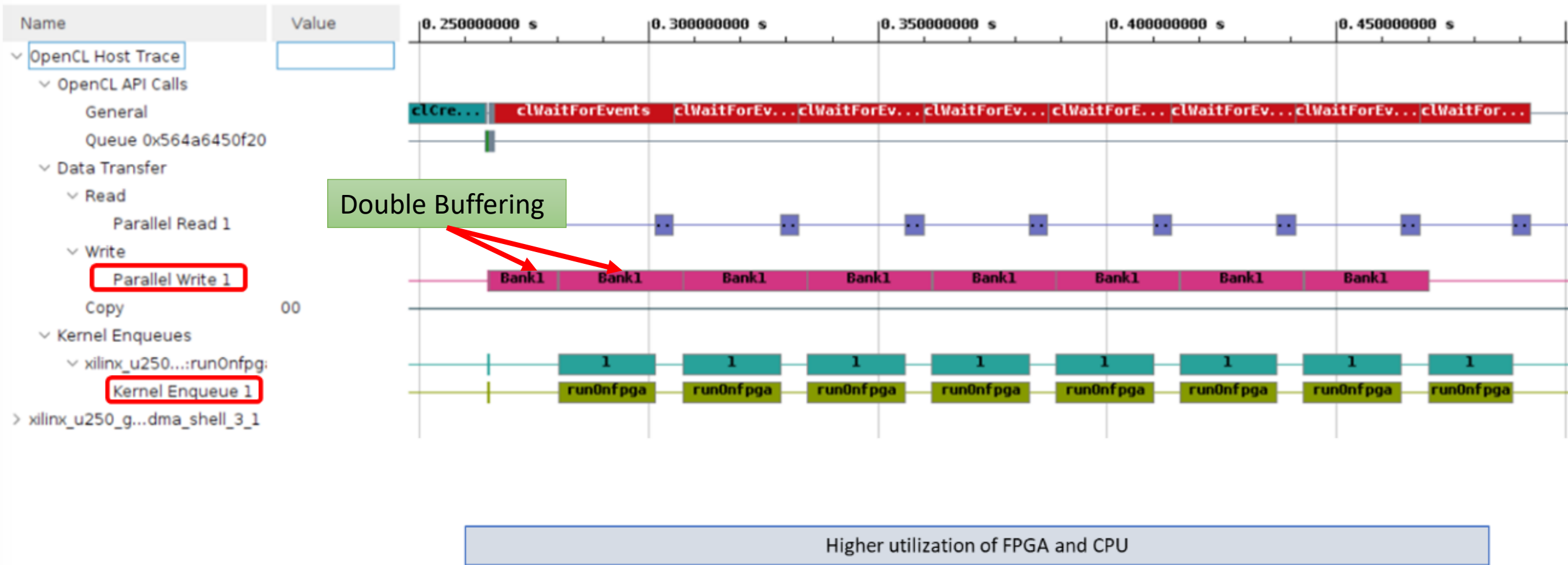
Out-of-order command queue



X23163-102620



Figure: Improved Timeline



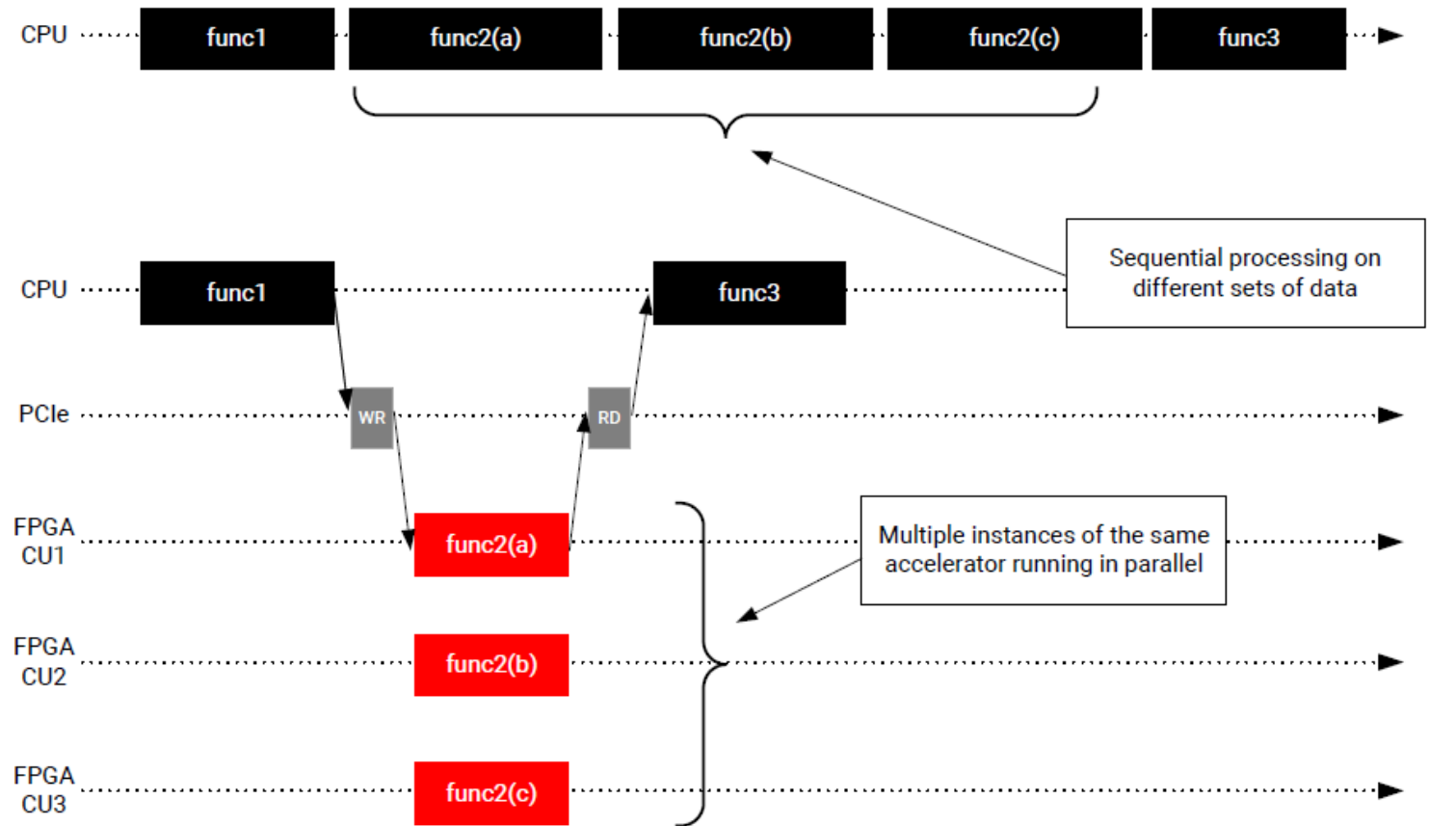
# Kernel Parallelism

# Identify Kernel Parallelism

- Wider datapath and processing more data samples in parallel (SIMD)
- Multiple Kernels (CUs)
- Task-level parallelism

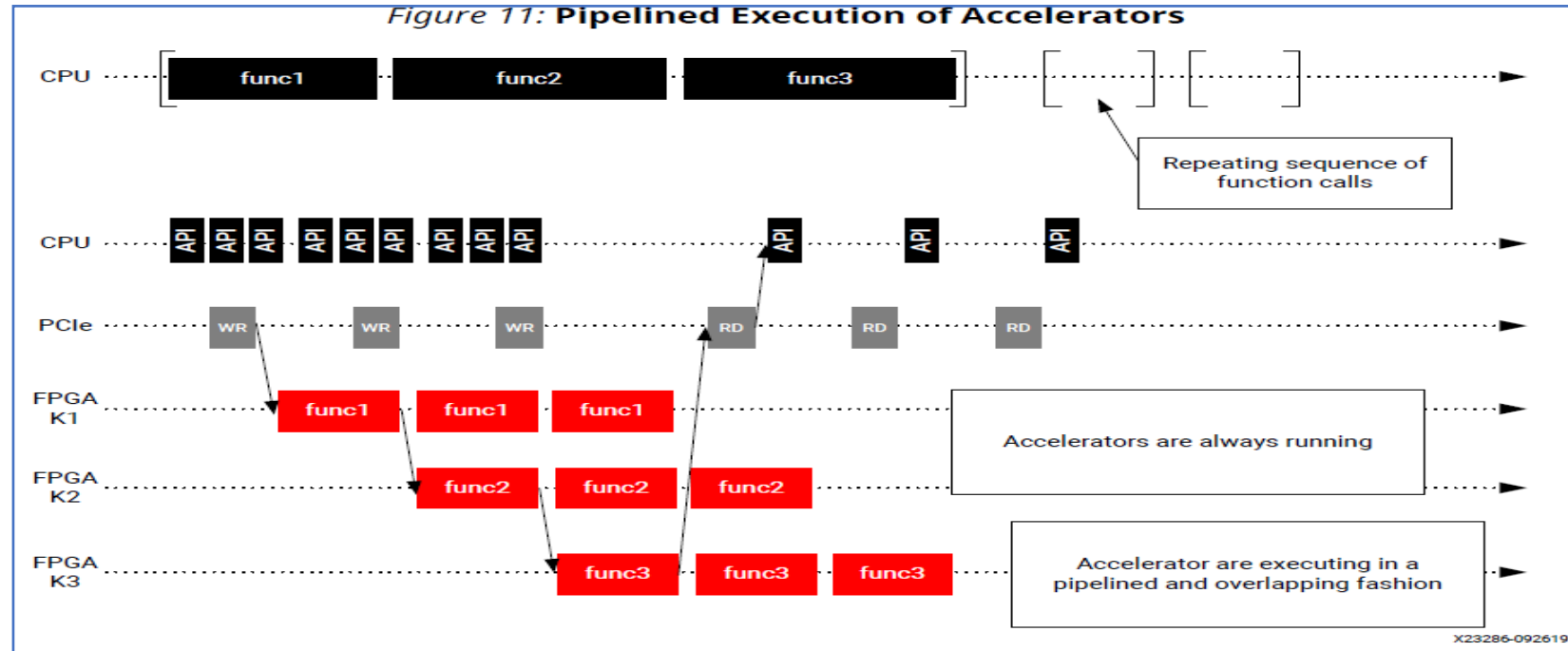
# Use Multiple Compute Units for Data Parallelism

Figure 9: Improving Performance with Multiple Compute Units



X23284-092619

# Use different current kernel for task-level parallelism

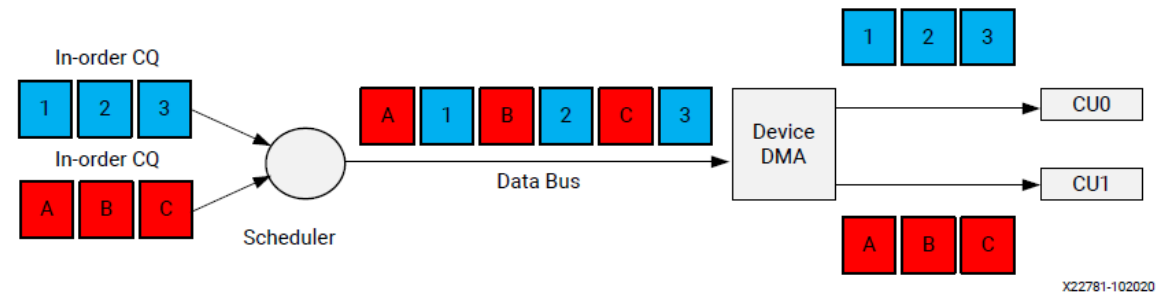


- Kernels use dataflow, instruction-level parallelism
- Data not written to Global Memory
- Host does not need to program kernel (ap\_ctrl\_none)
- Auto-restart kernel to save host API call.

# Kernel Scheduling for multiple CUs (Out-of-Order)

- CQ0, CQ1 in any order
- Commands in CQ are in order

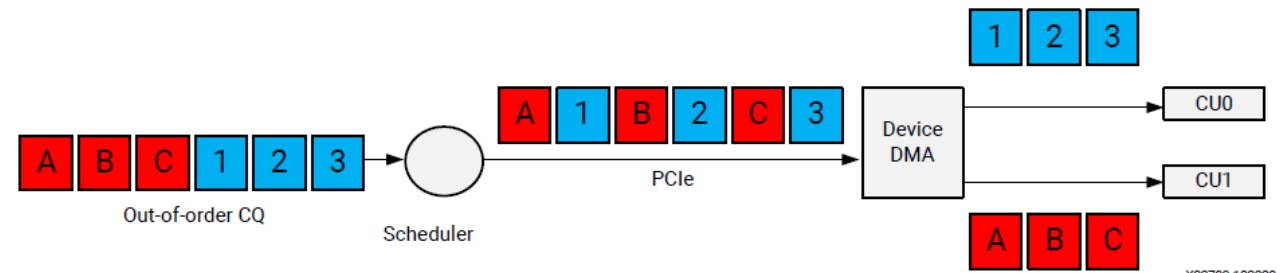
Figure 44: Example with Two In-Order Command Queues



X22781-102020

- Scheduler dispatch commands in any order subject to its resource availability
- Must manually define event dependency and synchronization

Figure 45: Example with Single Out-of-Order Command Queue



X22783-102020

**Tip: Using a single out-of-order command queue and manage event dependencies and synchronization explicitly, instead of using multiple command queues**

# Optimizing Compute and Transfer

- When data block is too large, kernel waits until all data transferred.
- Subdivide the buffer to balance the execution and transfer time
- Enqueue multiple runs of the kernels

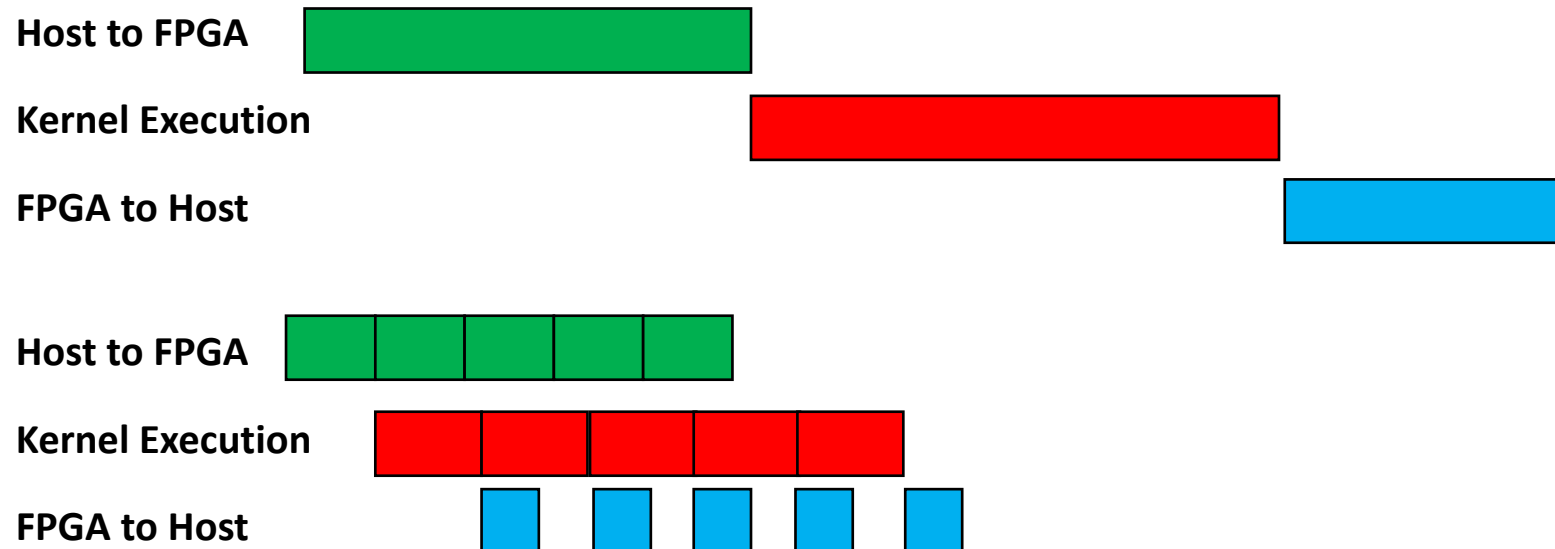


Table 2. Frequently Used Optimization Pragas in Vitis HLS

Pragma	Description
pragma HLS dataflow	Enables task-level pipelining to overlap functions and loops with each other
pragma HLS dependence	Provides additional loop dependency information for parallelization
pragma HLS loop_flatten	Flattens a nested loop to a single loop hierarchy with improved latency
pragma HLS loop_merge	Merges consecutive loops into a single loop to reduce latency and resource usage
pragma HLS occurrence	Allows less-executed operations to be pipelined at a slower rate
pragma HLS pipeline	Enables instruction-level pipelining to allow concurrent operation execution
pragma HLS unroll	Creates multiple copies of the loop body for parallelization of the iterations
pragma HLS aggregate	Collects and groups the data fields of a struct into a single scalar
pragma HLS array_partition	Splits an array into multiple entities for parallel access
pragma HLS array_reshap	Merges multiple elements in an array into one element for parallel access
pragma HLS disaggregate	Deconstructs the data fields of a struct into multiple scalars
pragma HLS stream	Implements a variable as a FIFO and specifies the depth
pragma HLS function_instantiate	Creates task-specific modules to handle specific conditions of a function call
pragma HLS inline	Removes a function as a separate entity in the hierarchy
pragma HLS interface	Specifies the implementation protocol of a function's ports
pragma HLS allocation	Limits the resource allocation in the implemented RTL
pragma HLS bind_op	Specifies the implementation of operations in the code
pragma HLS bind_storage	Specifies the memory type of a variable in the code
pragma HLS expression_balance	Rearranges operations with associative and commutative laws to reduce latency
pragma HLS latency	Specifies the minimum and/or maximum latency for the completion of a region