High Level Synthesis

High Level Synthesis

In one word:
The High-Level Synthesis (HLS) tool transforms a C specification into an RTL implementation that you can synthesize into a Xilinx FPGA.

Benefits:
-- Working at a higher level of abstraction while creating high-performance hardware.
-- Accelerating the computationally intensive parts of their algorithms on FPGA.

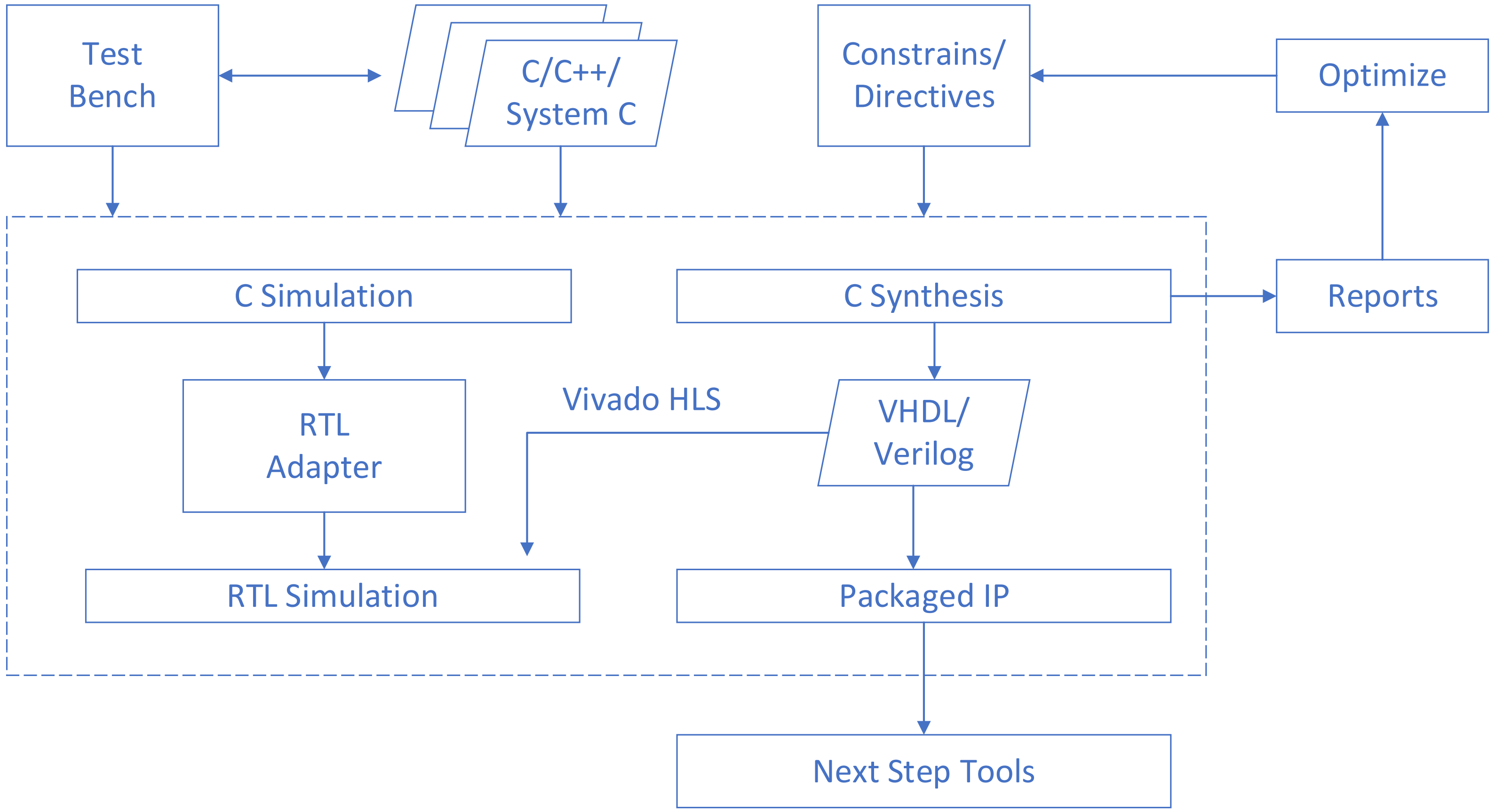
Using a high-level synthesis design methodology allows you to:
-- Develop algorithms at the C-level.
-- Verify at the C-level.
-- Control the C synthesis process through optimization directives.
-- Create multiple implementations from the C source code using optimization directives.

High Level Synthesis

High-level synthesis synthesizes the C code as follows:
-- Top-level function arguments synthesize into RTL I/O ports.
-- C functions synthesize into blocks in the RTL hierarchy.
-- Loops in the C functions are kept rolled by default.
-- Arrays in the C code synthesize into block RAM in the final FPGA design.

You can use optimization directives to modify and control the default behavior of the internal logic and I/O ports. This allows you to generate variations of the hardware implementation from the same C code.

Vivado HLS Design Flow



HLS : Reports

-- **Area**: Amount of hardware resources required to implement the design based on the resources available in the FPGA, including look-up tables (LUT), registers, block RAMs, and DSP48s.

-- **Latency**: Number of clock cycles required for the function to compute all output values.

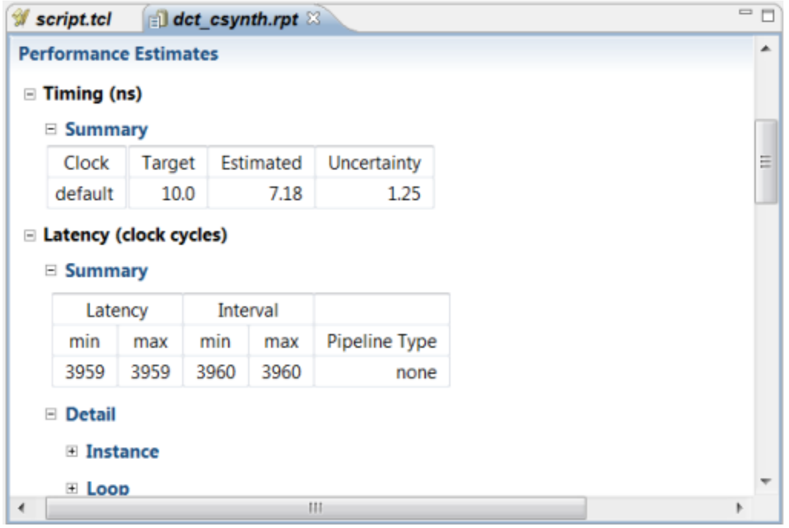
-- **Initiation interval (II)**: Number of clock cycles before the function can accept new input data.

-- **Loop iteration latency**: Number of clock cycles it takes to complete one iteration of the loop.

-- **Loop initiation interval**: Number of clock cycles before the next iteration of the loop starts to process data.

-- **Loop latency**: Number of cycles to execute all iterations of the loop.

HLS : Reports



script.tcl | dct_csynth.rpt

Performance Estimates

⊟ **Timing (ns)**

⊟ **Summary**

Clock	Target	Estimated	Uncertainty
default	10.0	7.18	1.25

⊟ **Latency (clock cycles)**

⊟ **Summary**

Latency		Interval		
min	max	min	max	Pipeline Type
3959	3959	3960	3960	none

⊟ **Detail**

⊞ **Instance**

⊞ **Loop**

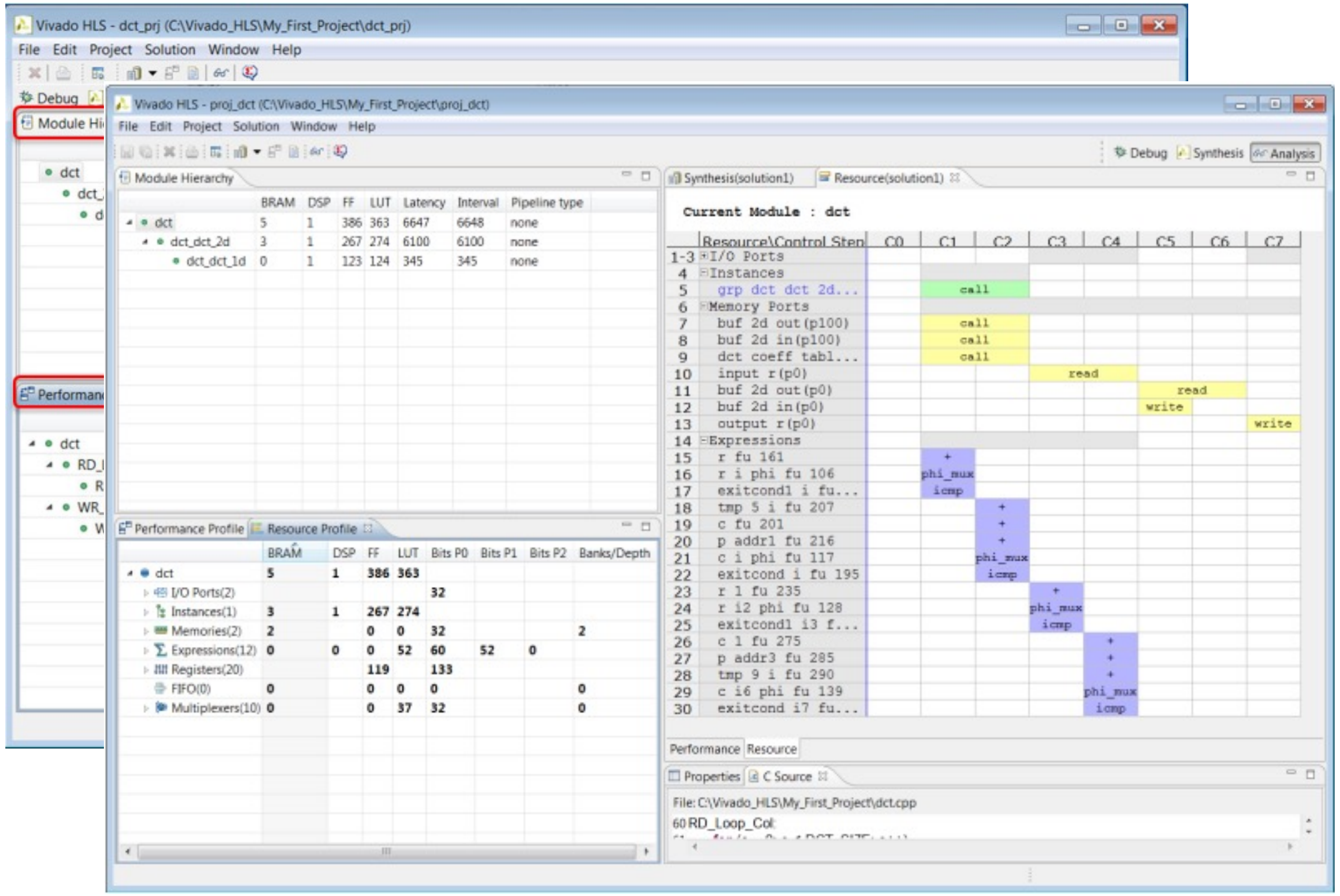
Synthesis Report

-- General Information

-- Performance Estimate

-- Utilization Estimate

-- Interface



Vivado HLS - dct_prj (C:\Vivado_HLS\My_First_Project\dct_prj)

Vivado HLS - proj_dct (C:\Vivado_HLS\My_First_Project\proj_dct)

Module Hierarchy

	BRAM	DSP	FF	LUT	Latency	Interval	Pipeline type
dct	5	1	386	363	6647	6648	none
dct_dct_2d	3	1	267	274	6100	6100	none
dct_dct_1d	0	1	123	124	345	345	none

Performance Profile | Resource Profile

	BRAM	DSP	FF	LUT	Bits P0	Bits P1	Bits P2	Banks/Depth
dct	5	1	386	363				
I/O Ports(2)					32			
Instances(1)	3	1	267	274				
Memories(2)	2		0	0	32			2
Expressions(12)	0	0	0	52	60	52	0	
Registers(20)			119		133			
FIFO(0)	0		0	0	0			0
Multiplexers(10)	0		0	37	32			0

Synthesis(solution1) | Resource(solution1)

Current Module : dct

	Resource\Control Step	C0	C1	C2	C3	C4	C5	C6	C7
1-3	⊞I/O Ports								
4	⊟Instances								
5	grp dct dct 2d...		call						
6	⊟Memory Ports								
7	buf 2d out(p100)		call						
8	buf 2d in(p100)		call						
9	dct coeff tabl...		call						
10	input r(p0)				read				
11	buf 2d out(p0)						read		
12	buf 2d in(p0)						write		
13	output r(p0)								write
14	⊟Expressions								
15	r fu 161		+						
16	r i phi fu 106		phi_mux						
17	exitcond1 i fu...		icmp						
18	tmp 5 i fu 207			+					
19	c fu 201			+					
20	p addr1 fu 216			+					
21	c i phi fu 117			phi_mux					
22	exitcond i fu 195			icmp					
23	r 1 fu 235				+				
24	r i2 phi fu 128				phi_mux				
25	exitcond1 i3 f...				icmp				
26	c 1 fu 275					+			
27	p addr3 fu 285					+			
28	tmp 9 i fu 290					+			
29	c i6 phi fu 139					phi_mux			
30	exitcond i7 fu...					icmp			

Performance | Resource

Properties | C Source

File: C:\Vivado_HLS\My_First_Project\dct.cpp

```
60 RD_Loop_Col:
```

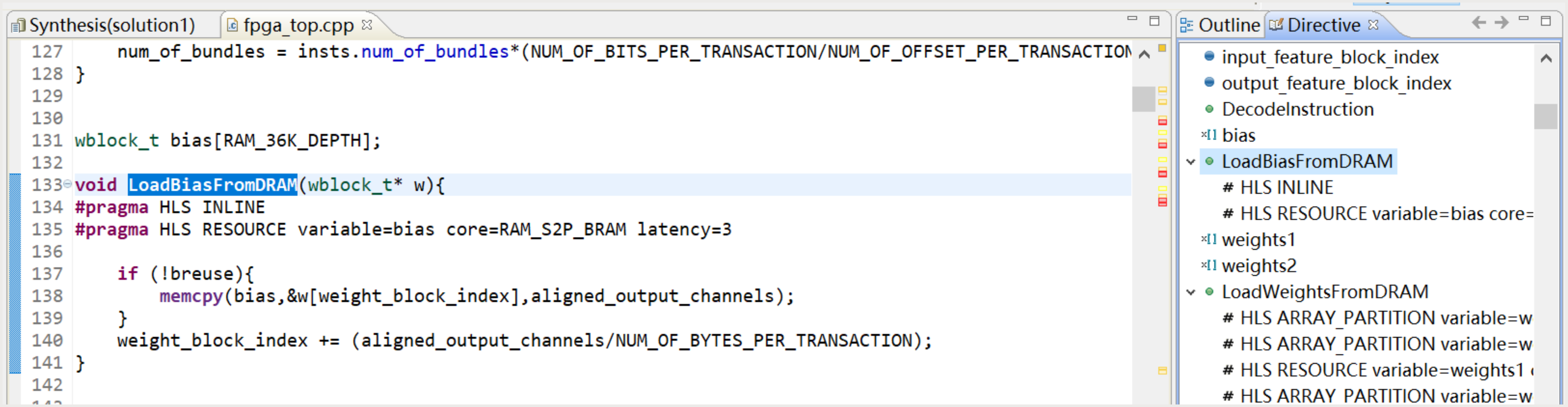
Analysis Perspective

HLS : Optimize

-- Instruct a task to execute in a pipeline, allowing the next execution of the task to begin before the current execution is complete.
-- Specify a latency for the completion of functions, loops, and regions.
-- Specify a limit on the number of resources used.
-- Override the inherent or implied dependencies in the code and permit specified operations. For example, if it is acceptable to discard or ignore the initial data values, such as in a video stream, allow a memory read before write if it results in better performance.
-- Select the I/O protocol to ensure the final design can be connected to other hardware blocks with the same I/O protocol.

Applying Optimization Directives

- You can add optimization directives via directives.tcl/ GUI/ source code.
- With the source code active in the Information pane, select the Directives tab on the right to display and modify directives for the file. The Directives tab contains all the objects and scopes in the currently opened source code to which you can apply directives.



```
    num_of_bundles = insts.num_of_bundles*(NUM_OF_BITS_PER_TRANSACTION/NUM_OF_OFFSET_PER_TRANSACTION
}


wblock_t bias[RAM_36K_DEPTH];

void LoadBiasFromDRAM(wblock_t* w){
#pragma HLS INLINE
#pragma HLS RESOURCE variable=bias core=RAM_S2P_BRAM latency=3

    if (!breuse){
        memcpy(bias,&w[weight_block_index],aligned_output_channels);
    }
    weight_block_index += (aligned_output_channels/NUM_OF_BYTES_PER_TRANSACTION);
}
```

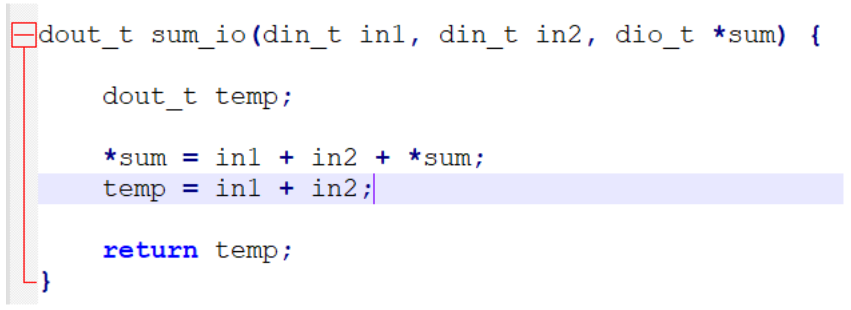
```
input_feature_block_index
output_feature_block_index
DecodeInstruction
bias
LoadBiasFromDRAM
    # HLS INLINE
    # HLS RESOURCE variable=bias core=
weights1
weights2
LoadWeightsFromDRAM
    # HLS ARRAY_PARTITION variable=w
    # HLS ARRAY_PARTITION variable=w
    # HLS RESOURCE variable=weights1 
    # HLS ARRAY_PARTITION variable=w
```

Applying Optimization Directives

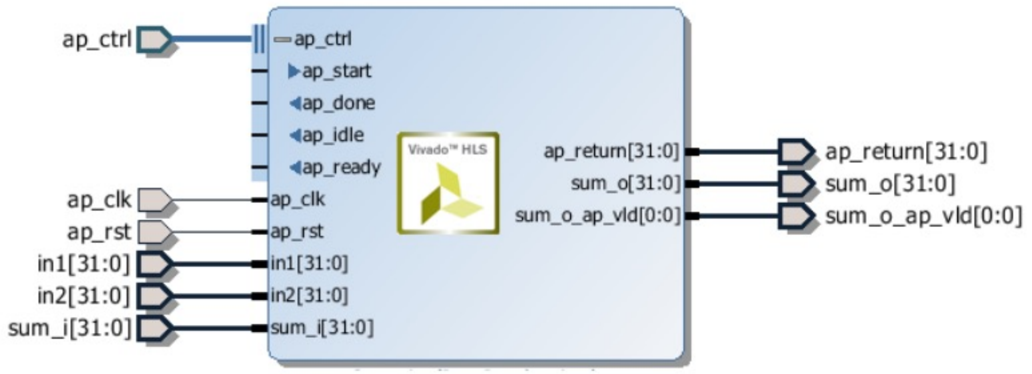
You can apply optimization directives to the following objects and scopes:
-- **Interfaces**: When you apply directives to an interface, Vivado HLS applies the directive to the top-level function, because the top-level function is the scope that contains the interface.
-- **Functions**: When you apply directives to functions, Vivado HLS applies the directive to all objects within the scope of the function. The effect of any directive stops at the next level of function hierarchy.
-- **Loops**: When you apply directives to loops, Vivado HLS applies the directive to all objects within the scope of the loop.
-- **Arrays**: When you apply directives to arrays, Vivado HLS applies the directive to the scope that contains the array.
-- **Regions**: When you apply directives to regions, Vivado HLS applies the directive to the entire scope of the region. A region is any area enclosed within two braces.

Interface Synthesis

When the top-level function is synthesized, the arguments (or parameters) to the function are synthesized into RTL ports. This process is called *interface synthesis*.



```
dout_t sum_io(din_t in1, din_t in2, dio_t *sum) {

    dout_t temp;

    *sum = in1 + in2 + *sum;
    temp = in1 + in2;

    return temp;
}
```



-- Two inputs in1 and in2.

-- A pointer sum that is both read from and written to.

-- Return value temp.

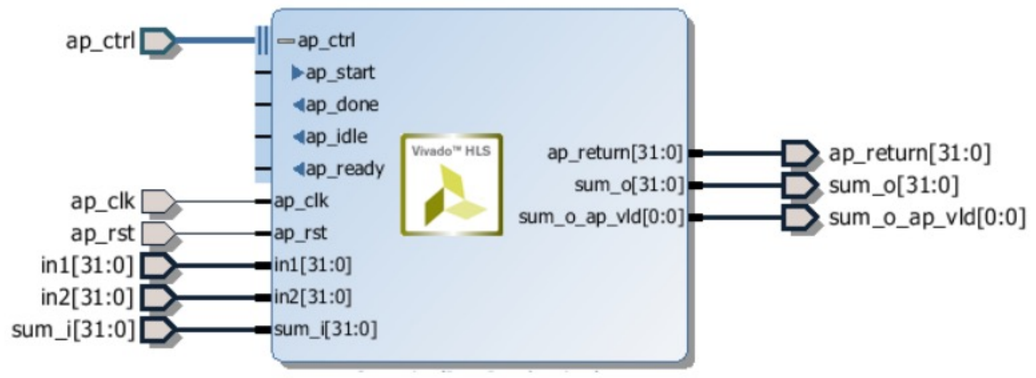
-- Clock and Reset ports.

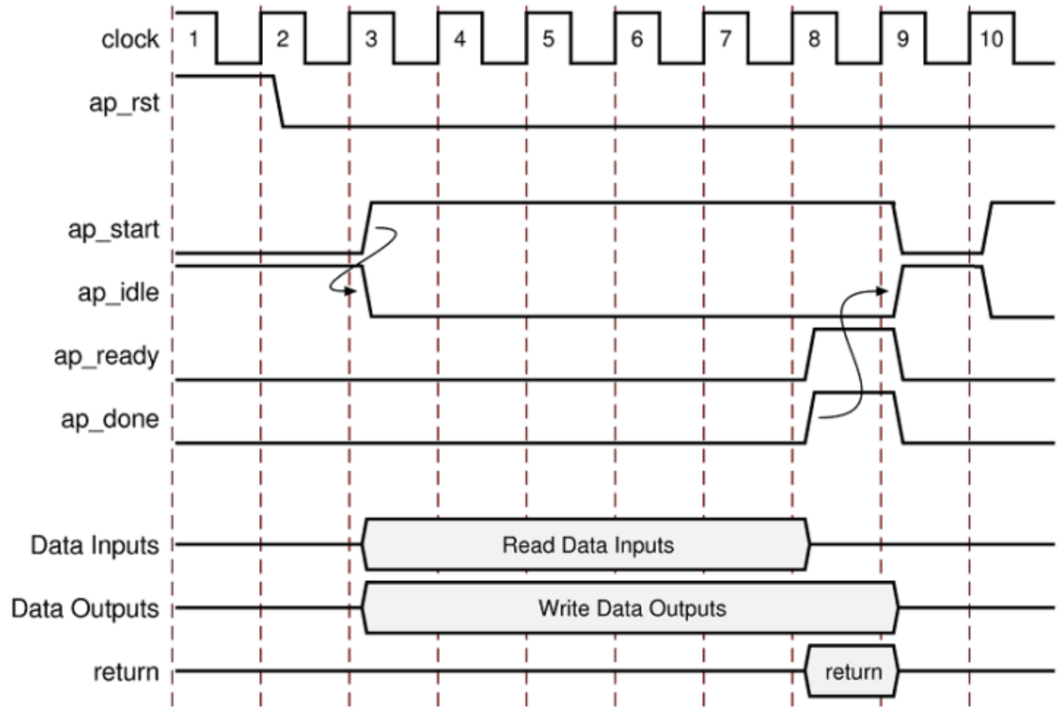
-- Block level interface protocol.

-- Port level interface protocol.

Interface Synthesis

When the top-level function is synthesized, the arguments (or parameters) to the function are synthesized into RTL ports. This process is called *interface synthesis*.





-- Clock and Reset ports.

-- Block level interface protocol.

-- Port level interface protocol.

Interface Synthesis

The interface MODE option has types below:

-- **ap_fifo**: Implements the port with a standard FIFO interface using data input and output ports with associated active-Low FIFO empty and full ports.

-- **ap_bus**: Implements pointer and pass-by-reference ports as a bus interface.

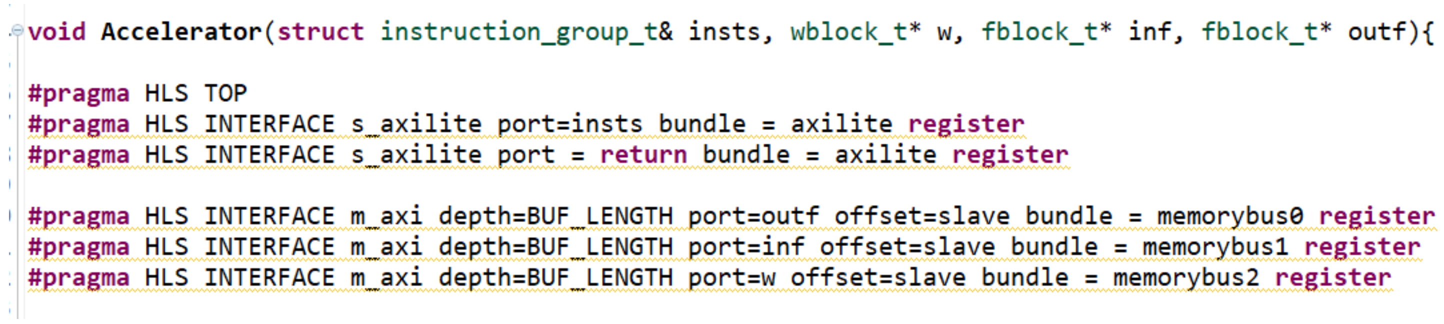
-- **ap_memory**: Implements array arguments as a standard RAM interface.

-- **bram**: Implements array arguments as a standard RAM interface. If you use the RTL design in Vivado IP integrator, the memory interface appears as a single port.

-- **axis**: Implements all ports as an AXI4-Stream interface.

-- **s_axilite**: Implements all ports as an AXI4-Lite interface.

-- **m_axi**: Implements all ports as an AXI4 interface.



```
void Accelerator(struct instruction_group_t& insts, wblock_t* w, fblock_t* inf, fblock_t* outf){

#pragma HLS TOP
#pragma HLS INTERFACE s_axilite port=insts bundle = axilite register
#pragma HLS INTERFACE s_axilite port = return bundle = axilite register

#pragma HLS INTERFACE m_axi depth=BUF_LENGTH port=outf offset=slave bundle = memorybus0 register
#pragma HLS INTERFACE m_axi depth=BUF_LENGTH port=inf offset=slave bundle = memorybus1 register
#pragma HLS INTERFACE m_axi depth=BUF_LENGTH port=w offset=slave bundle = memorybus2 register
```

Optimize -- Clocks, resets and RTL Outputs

Clocks:
-- For C and C++ designs only a single clock is supported. The same clock is applied to all functions in the design.
For SystemC designs, each SC_MODULE may be specified with a different clock.

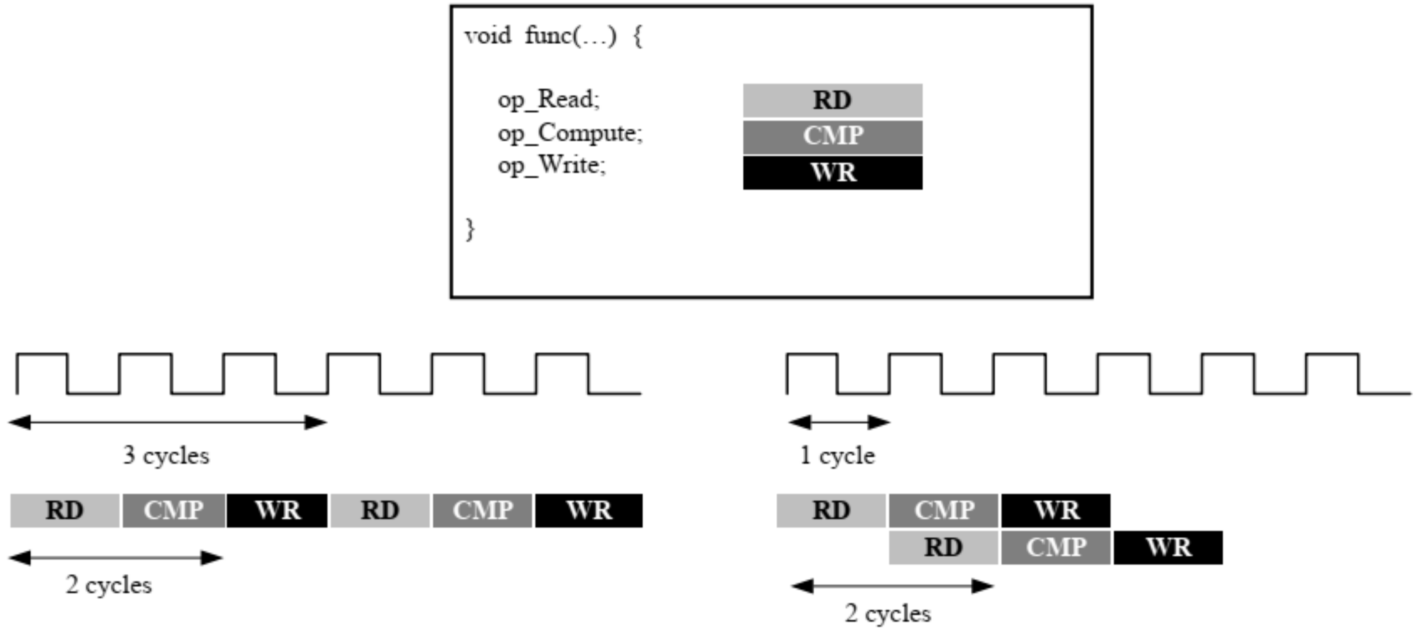
Reset:
-- None, no reset is added to the design.
-- Control, the default, all control registers are reset.
-- State, this adds a reset to control registers plus any registers or memories derived from static and global variables in the C code.
-- All, to add a reset to all registers and memories in the design.

RTL Outputs:
-- Specify the type of FSM encoding used in the RTL state machines.
-- Add an arbitrary comment string, such as a copyright notice to all RTL files.
-- Specify a unique name to all RTL output file names.
-- Force the RTL ports to use lower names.

Optimizing for Throughput

Function and Loop Pipelining:

- Pipelining allows operations to happen concurrently: each execution step does not have to complete all operations before it begins the next operation.
- Pipelining is applied to functions and loops.



```
void func(...) {

    op_Read;
    op_Compute;
    op_Write;

}
```

(a) Without Function Pipelining

(b)With Function Pipelining

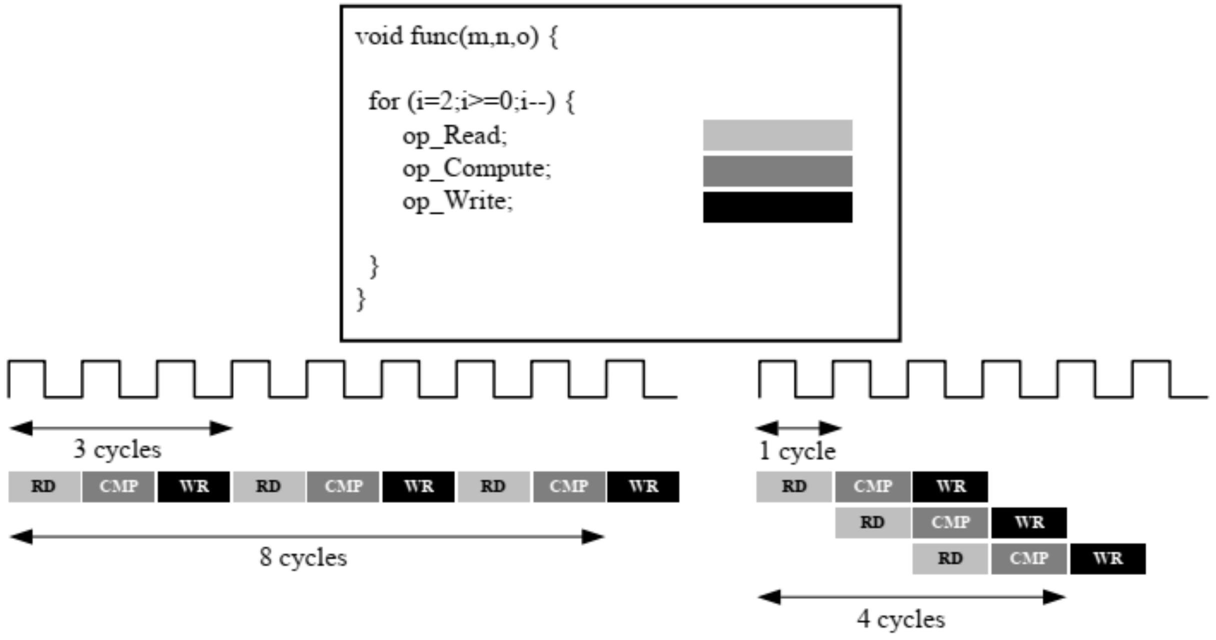
-- Without Function Pipelining, the function has an initiation interval (II) of 3 and a latency of 3.

-- With Function Pipelining, a new input is read every cycle (II=1) with no change to the output latency.

Optimizing for Throughput

Function and Loop Pipelining:

- Pipelining allows operations to happen concurrently: each execution step does not have to complete all operations before it begins the next operation.
- Pipelining is applied to functions and loops.



```
void func(m,n,o) {

  for (i=2;i>=0;i--) {
      op_Read;
      op_Compute;
      op_Write;

  }
}
```

(a) Without Loop Pipelining (b)With Loop Pipelining

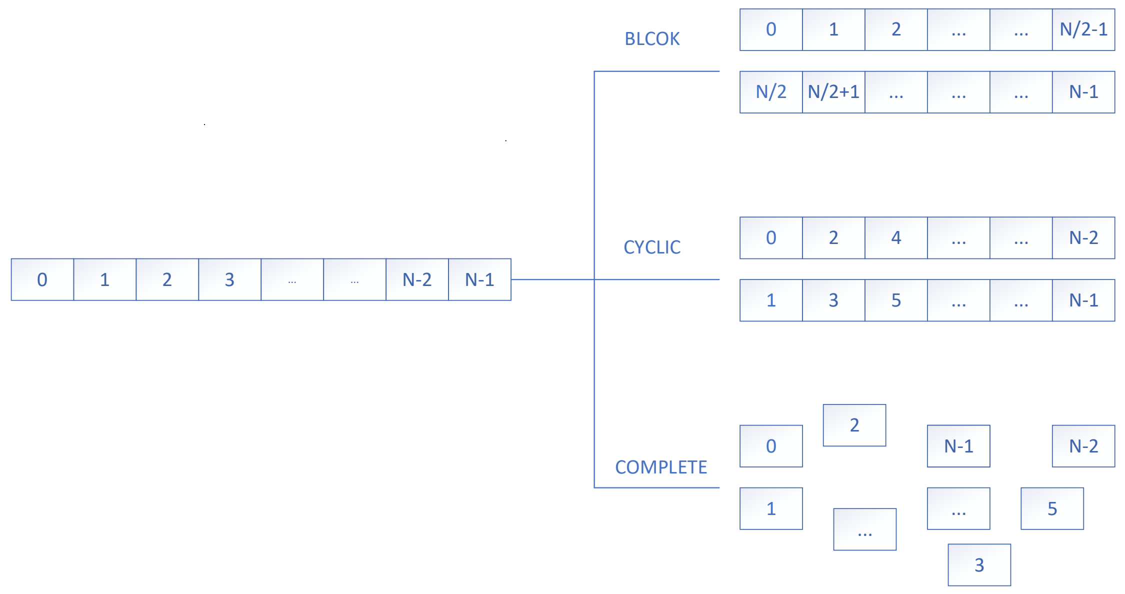
-- In (a), there are 3 clock cycles between each input read (II=3), and it requires 8 clock cycles before the last output write is performed.

-- In (b), a new input sample is read every cycle (II=1) and the final output is written after only 4 clock cycles.

Optimize – Partitioning Arrays

Warning： Unable to schedule ‘xxx' operation on array 'mem' due to limited memory ports.

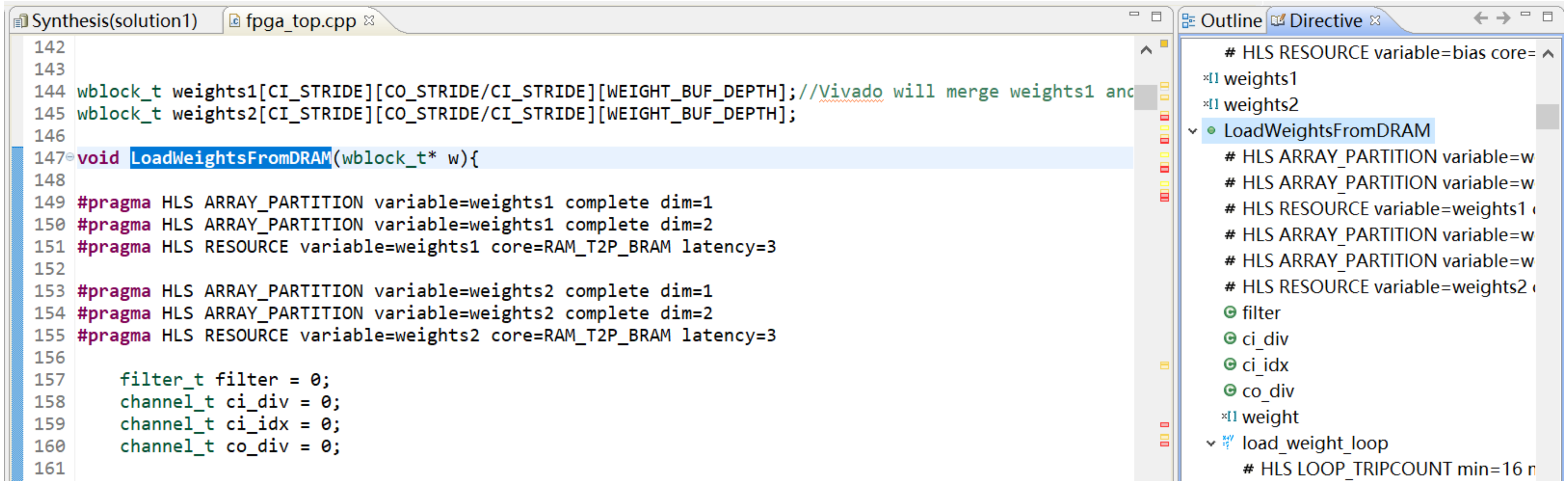
-- Arrays are implemented as **block RAM** which only has a maximum of two data ports.
-- The issue above can be solved by splitting the array (a single block RAM resource) into multiple smaller arrays (multiple block RAMs), effectively increasing the number of ports.



The original array is split into equally sized blocks of consecutive elements of the original array.

The original array is split into equally sized blocks interleaving the elements of the original array.

The default operation is to split the array into its individual elements. This
corresponds to resolving a memory into registers.

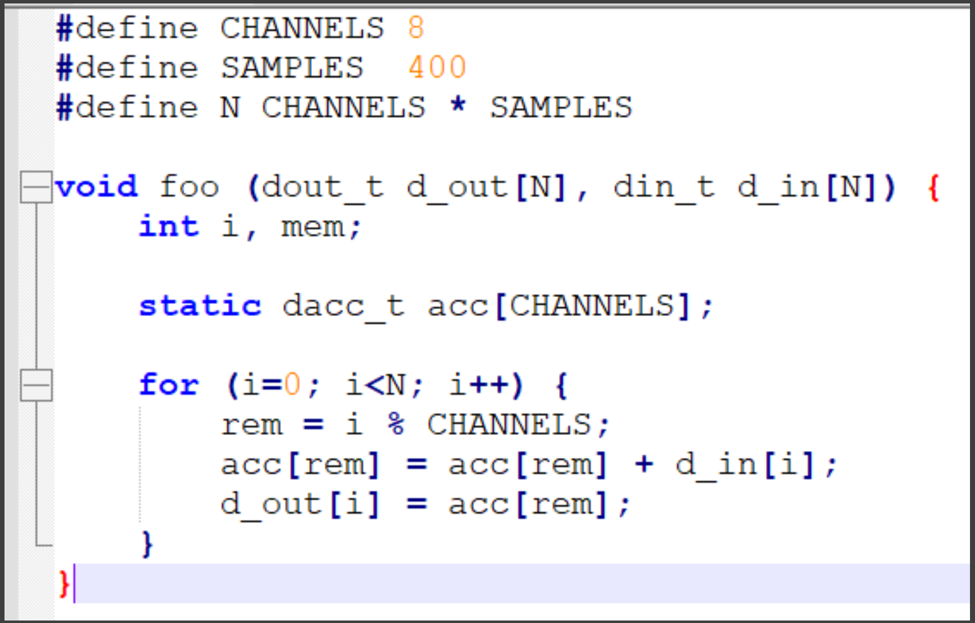


```

wblock_t weights1[CI_STRIDE][CO_STRIDE/CI_STRIDE][WEIGHT_BUF_DEPTH];//Vivado will merge weights1 and
wblock_t weights2[CI_STRIDE][CO_STRIDE/CI_STRIDE][WEIGHT_BUF_DEPTH];

void LoadWeightsFromDRAM(wblock_t* w){

#pragma HLS ARRAY_PARTITION variable=weights1 complete dim=1
#pragma HLS ARRAY_PARTITION variable=weights1 complete dim=2
#pragma HLS RESOURCE variable=weights1 core=RAM_T2P_BRAM latency=3

#pragma HLS ARRAY_PARTITION variable=weights2 complete dim=1
#pragma HLS ARRAY_PARTITION variable=weights2 complete dim=2
#pragma HLS RESOURCE variable=weights2 core=RAM_T2P_BRAM latency=3

    filter_t filter = 0;
    channel_t ci_div = 0;
    channel_t ci_idx = 0;
    channel_t co_div = 0;

```

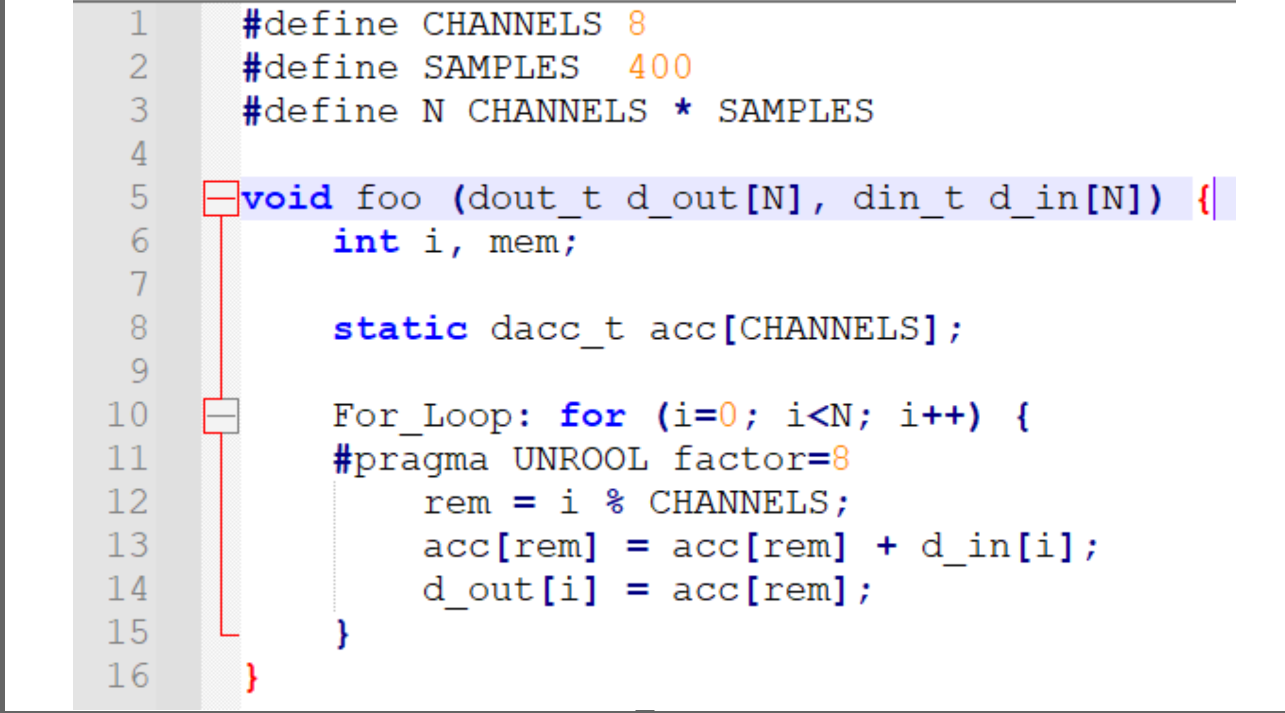
```
    # HLS RESOURCE variable=bias core=
  weights1
  weights2
  LoadWeightsFromDRAM
    # HLS ARRAY_PARTITION variable=w
    # HLS ARRAY_PARTITION variable=w
    # HLS RESOURCE variable=weights1
    # HLS ARRAY_PARTITION variable=w
    # HLS ARRAY_PARTITION variable=w
    # HLS RESOURCE variable=weights2
    filter
    ci_div
    ci_idx
    co_div
    weight
    load_weight_loop
      # HLS LOOP_TRIPCOUNT min=16 r
```

Optimize – Loop Unrolling

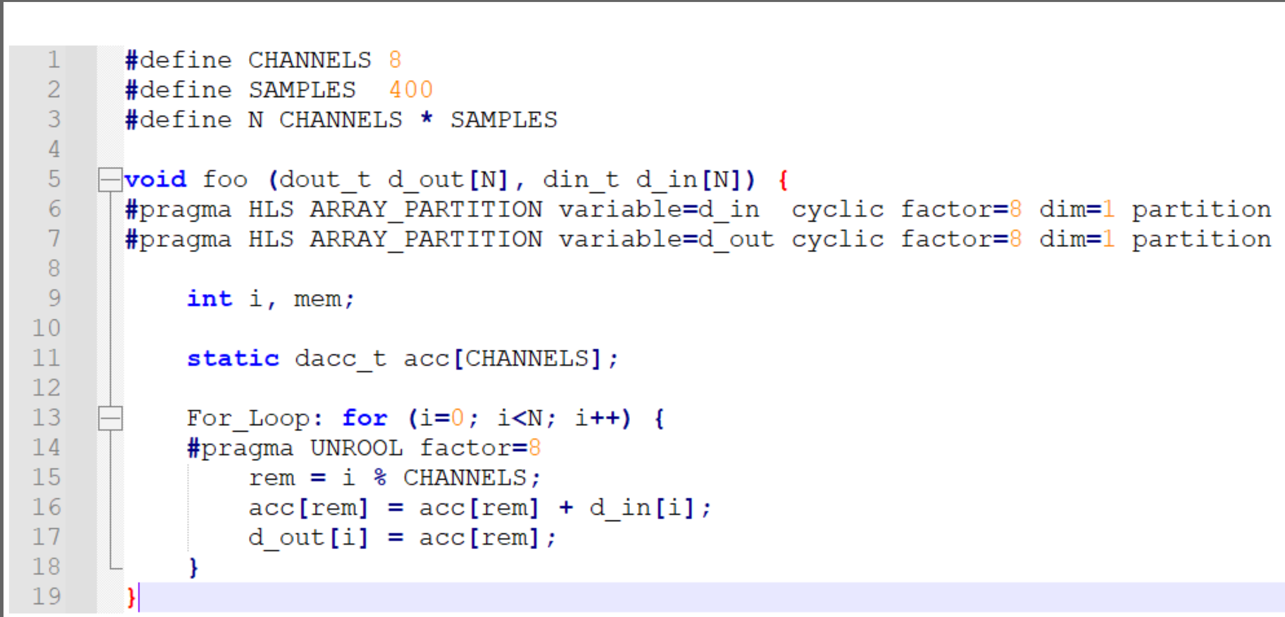
- By default, loops are kept rolled in Vivado HLS. We can use UNROLL directive to unroll the loop.



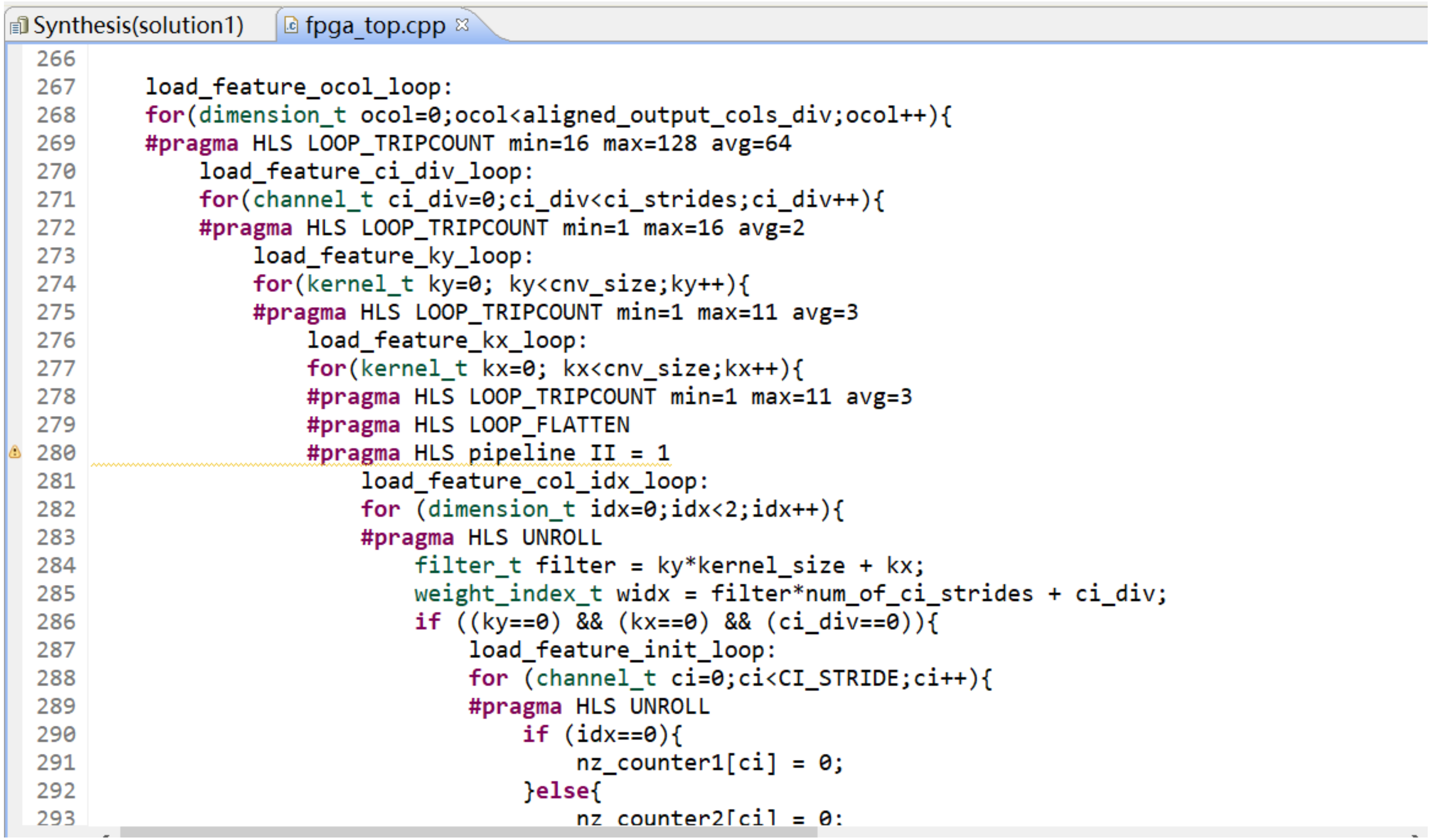
```
#define CHANNELS 8
#define SAMPLES  400
#define N CHANNELS * SAMPLES

void foo (dout_t d_out[N], din_t d_in[N]) {
    int i, mem;

    static dacc_t acc[CHANNELS];

    for (i=0; i<N; i++) {
        rem = i % CHANNELS;
        acc[rem] = acc[rem] + d_in[i];
        d_out[i] = acc[rem];
    }
}
```



```
#define CHANNELS 8
#define SAMPLES  400
#define N CHANNELS * SAMPLES

void foo (dout_t d_out[N], din_t d_in[N]) {
    int i, mem;

    static dacc_t acc[CHANNELS];

    For_Loop: for (i=0; i<N; i++) {
    #pragma UNROOL factor=8
        rem = i % CHANNELS;
        acc[rem] = acc[rem] + d_in[i];
        d_out[i] = acc[rem];
    }
}
```



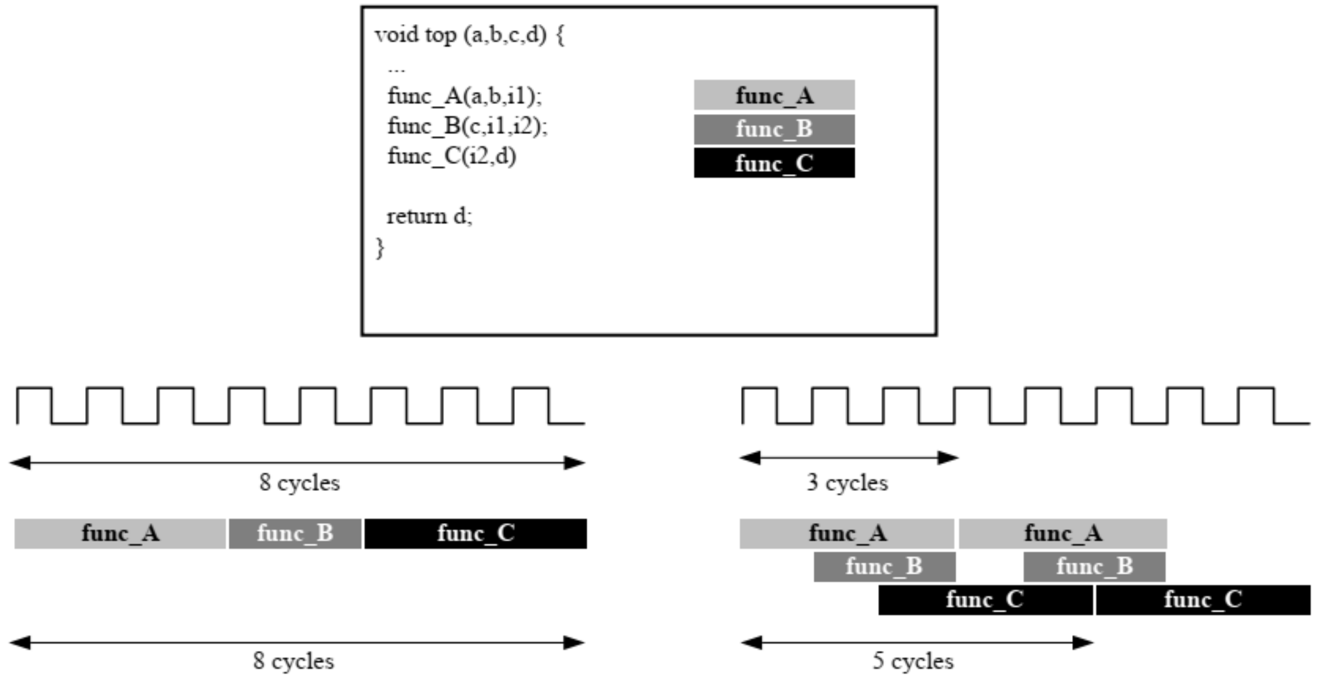
```
#define CHANNELS 8
#define SAMPLES  400
#define N CHANNELS * SAMPLES

void foo (dout_t d_out[N], din_t d_in[N]) {
#pragma HLS ARRAY_PARTITION variable=d_in  cyclic factor=8 dim=1 partition
#pragma HLS ARRAY_PARTITION variable=d_out cyclic factor=8 dim=1 partition

    int i, mem;

    static dacc_t acc[CHANNELS];

    For_Loop: for (i=0; i<N; i++) {
    #pragma UNROOL factor=8
        rem = i % CHANNELS;
        acc[rem] = acc[rem] + d_in[i];
        d_out[i] = acc[rem];
    }
}
```



```

    load_feature_ocol_loop:
    for(dimension_t ocol=0;ocol<aligned_output_cols_div;ocol++){
    #pragma HLS LOOP_TRIPCOUNT min=16 max=128 avg=64
        load_feature_ci_div_loop:
        for(channel_t ci_div=0;ci_div<ci_strides;ci_div++){
        #pragma HLS LOOP_TRIPCOUNT min=1 max=16 avg=2
            load_feature_ky_loop:
            for(kernel_t ky=0; ky<cnv_size;ky++){
            #pragma HLS LOOP_TRIPCOUNT min=1 max=11 avg=3
                load_feature_kx_loop:
                for(kernel_t kx=0; kx<cnv_size;kx++){
                #pragma HLS LOOP_TRIPCOUNT min=1 max=11 avg=3
                #pragma HLS LOOP_FLATTEN
                #pragma HLS pipeline II = 1
                    load_feature_col_idx_loop:
                    for (dimension_t idx=0;idx<2;idx++){
                    #pragma HLS UNROLL
                        filter_t filter = ky*kernel_size + kx;
                        weight_index_t widx = filter*num_of_ci_strides + ci_div;
                        if ((ky==0) && (kx==0) && (ci_div==0)){
                            load_feature_init_loop:
                            for (channel_t ci=0;ci<CI_STRIDE;ci++){
                            #pragma HLS UNROLL
                                if (idx==0){
                                    nz_counter1[ci] = 0;
                                }else{
                                    nz_counter2[ci] = 0;
```

Optimize – Dataflow

- The dataflow optimization is useful on a set of sequential tasks to create an architecture of concurrent processes.



```
void top (a,b,c,d) {
  ...
  func_A(a,b,i1);
  func_B(c,i1,i2);
  func_C(i2,d)

  return d;
}
```

(a) Without dataflow

(b) With dataflow

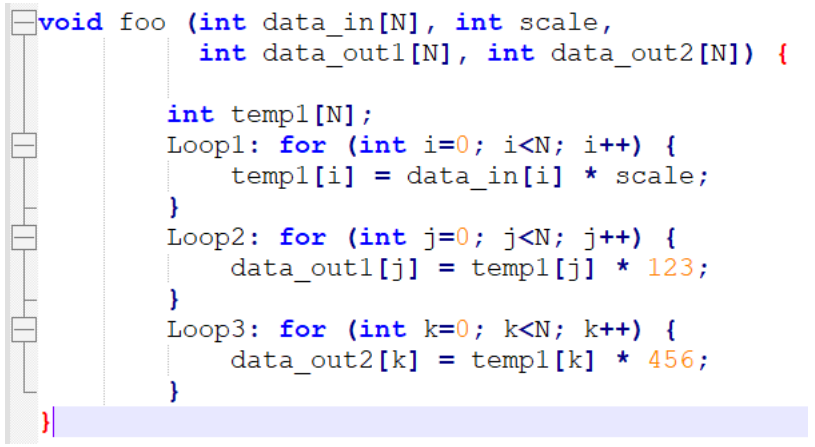
Dataflow Optimization Limitations

The following behaviors can prevent or limit the overlapping that Vivado HLS can perform with DATAFLOW optimization:

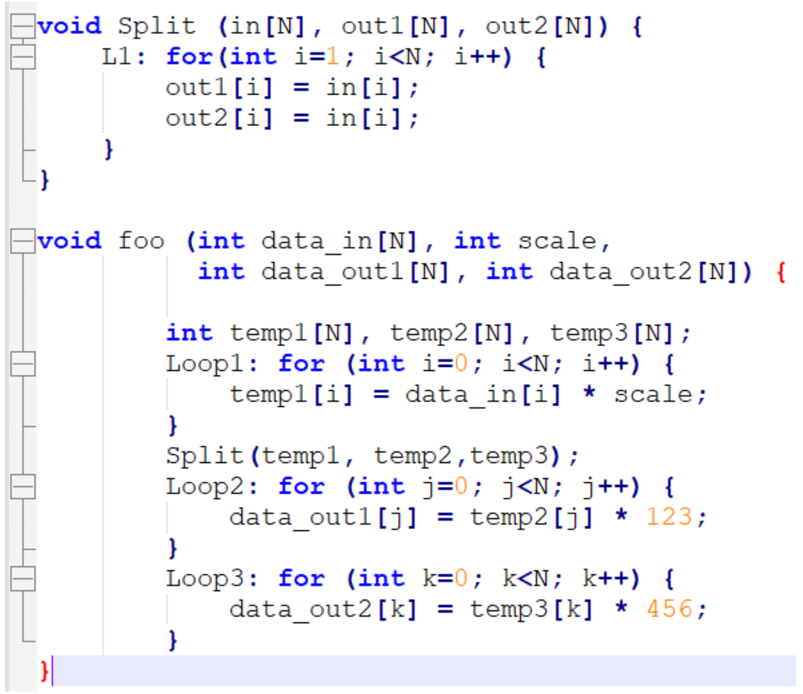
-- Single-producer-consumer violations
-- Bypassing tasks
-- Feedback between tasks
-- Conditional execution of tasks
-- Loops with multiple exit conditions

Dataflow Optimization Limitations

Single-producer-consumer Violations :
For Vivado HLS to perform the DATAFLOW optimization, all elements passed between tasks must follow a single-producer-consumer model. Each variable must be driven from a single task and only be consumed by a single task.



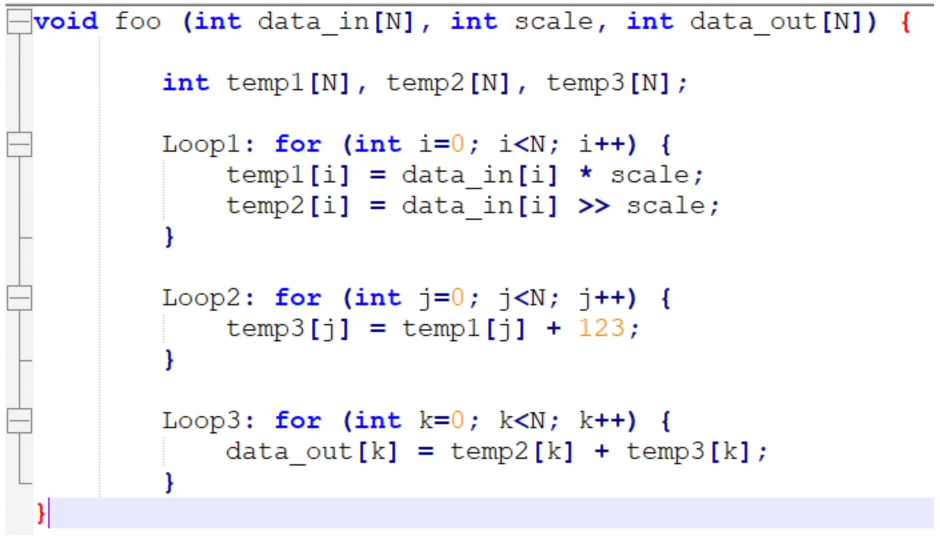
```
void foo (int data_in[N], int scale,
          int data_out1[N], int data_out2[N]) {

        int temp1[N];
        Loop1: for (int i=0; i<N; i++) {
            temp1[i] = data_in[i] * scale;
        }
        Loop2: for (int j=0; j<N; j++) {
            data_out1[j] = temp1[j] * 123;
        }
        Loop3: for (int k=0; k<N; k++) {
            data_out2[k] = temp1[k] * 456;
        }
}
```



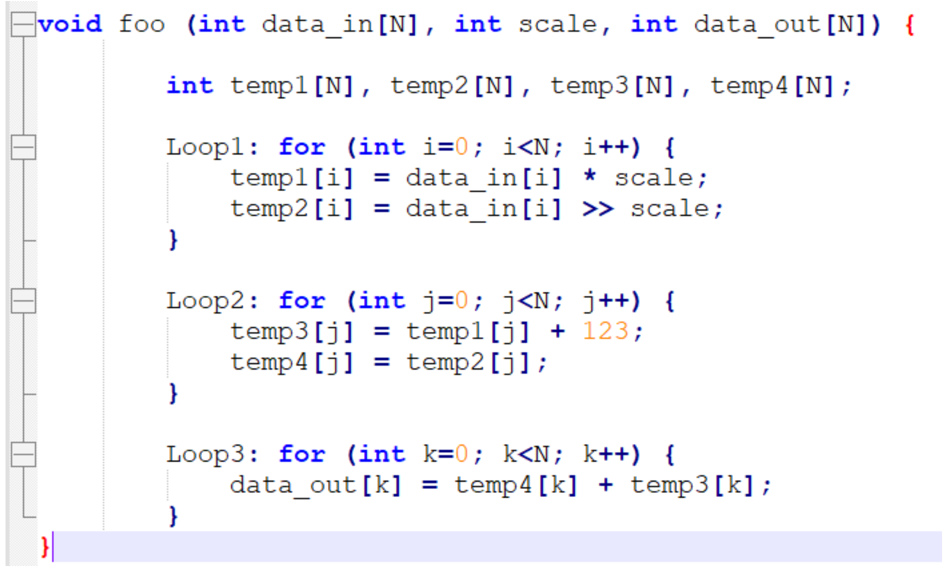
```
void Split (in[N], out1[N], out2[N]) {
    L1: for(int i=1; i<N; i++) {
        out1[i] = in[i];
        out2[i] = in[i];
    }
}

void foo (int data_in[N], int scale,
          int data_out1[N], int data_out2[N]) {

        int temp1[N], temp2[N], temp3[N];
        Loop1: for (int i=0; i<N; i++) {
            temp1[i] = data_in[i] * scale;
        }
        Split(temp1, temp2,temp3);
        Loop2: for (int j=0; j<N; j++) {
            data_out1[j] = temp2[j] * 123;
        }
        Loop3: for (int k=0; k<N; k++) {
            data_out2[k] = temp3[k] * 456;
        }
}
```

Dataflow Optimization Limitations

Bypassing Tasks:
Data should generally flow from one task to another. If you bypass tasks, this can reduce the performance of the DATAFLOW optimization.



```
void foo (int data_in[N], int scale, int data_out[N]) {

        int temp1[N], temp2[N], temp3[N];

        Loop1: for (int i=0; i<N; i++) {
            temp1[i] = data_in[i] * scale;
            temp2[i] = data_in[i] >> scale;
        }

        Loop2: for (int j=0; j<N; j++) {
            temp3[j] = temp1[j] + 123;
        }

        Loop3: for (int k=0; k<N; k++) {
            data_out[k] = temp2[k] + temp3[k];
        }
}
```

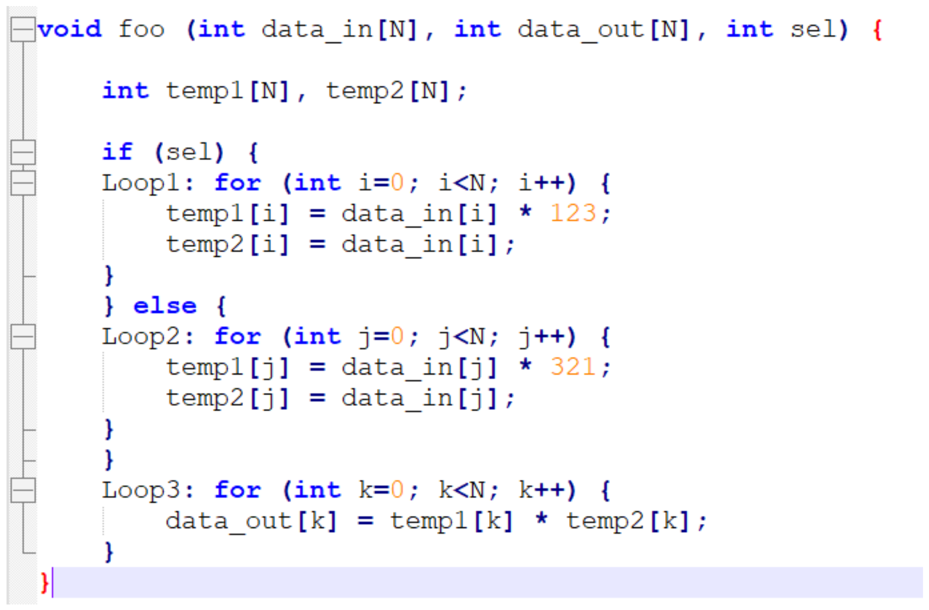


```
void foo (int data_in[N], int scale, int data_out[N]) {

        int temp1[N], temp2[N], temp3[N], temp4[N];

        Loop1: for (int i=0; i<N; i++) {
            temp1[i] = data_in[i] * scale;
            temp2[i] = data_in[i] >> scale;
        }

        Loop2: for (int j=0; j<N; j++) {
            temp3[j] = temp1[j] + 123;
            temp4[j] = temp2[j];
        }

        Loop3: for (int k=0; k<N; k++) {
            data_out[k] = temp4[k] + temp3[k];
        }
}
```

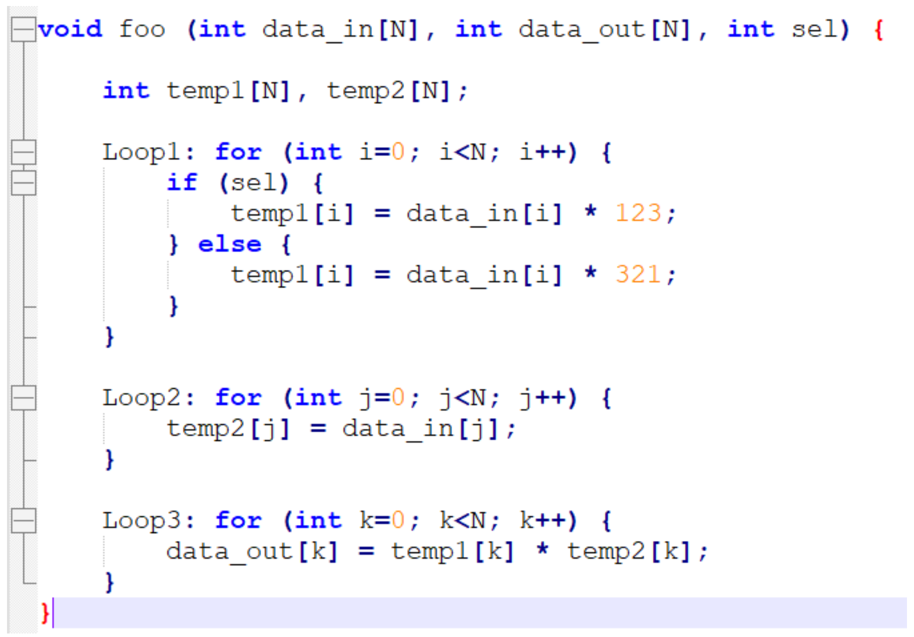
Dataflow Optimization Limitations

Conditional Execution of Tasks:

The DATAFLOW optimization does not optimize tasks that are conditionally executed. To ensure each loop is executed in all cases, you must make sure the loops are always executed, and data always flows from one loop to the next.



```
void foo (int data_in[N], int data_out[N], int sel) {

    int temp1[N], temp2[N];

    if (sel) {
    Loop1: for (int i=0; i<N; i++) {
        temp1[i] = data_in[i] * 123;
        temp2[i] = data_in[i];
    }
    } else {
    Loop2: for (int j=0; j<N; j++) {
        temp1[j] = data_in[j] * 321;
        temp2[j] = data_in[j];
    }
    }
    Loop3: for (int k=0; k<N; k++) {
        data_out[k] = temp1[k] * temp2[k];
    }
}
```



```
void foo (int data_in[N], int data_out[N], int sel) {

    int temp1[N], temp2[N];

    Loop1: for (int i=0; i<N; i++) {
        if (sel) {
            temp1[i] = data_in[i] * 123;
        } else {
            temp1[i] = data_in[i] * 321;
        }
    }

    Loop2: for (int j=0; j<N; j++) {
        temp2[j] = data_in[j];
    }

    Loop3: for (int k=0; k<N; k++) {
        data_out[k] = temp1[k] * temp2[k];
    }
}
```

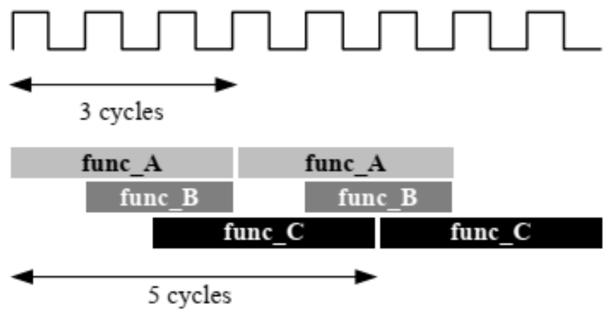
Dataflow Optimization Limitations

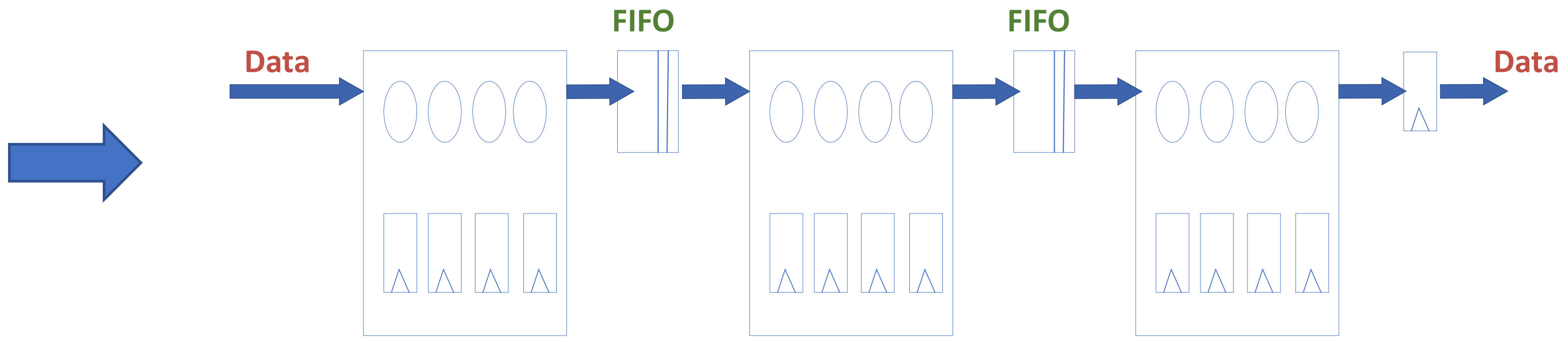
Feedback between Tasks:
Feedback occurs when the output from a task is consumed by a previous task in the DATAFLOW region. Feedback between tasks is not permitted in a DATAFLOW region.

Loops with Multiple Exit Conditions:
Loops with multiple exit points cannot be used in a DATAFLOW region.

Finally, the DATAFLOW optimization has no hierarchical implementation. If a sub-function or loop contains additional tasks that might benefit from the DATAFLOW optimization, you must apply the DATAFLOW optimization to the loop, the sub-function, or inline the sub-function.

Configuring DATAFLOW Memory Channels





-- For scalar, pointer, and reference parameters, Vivado HLS implements the channel as a FIFO.

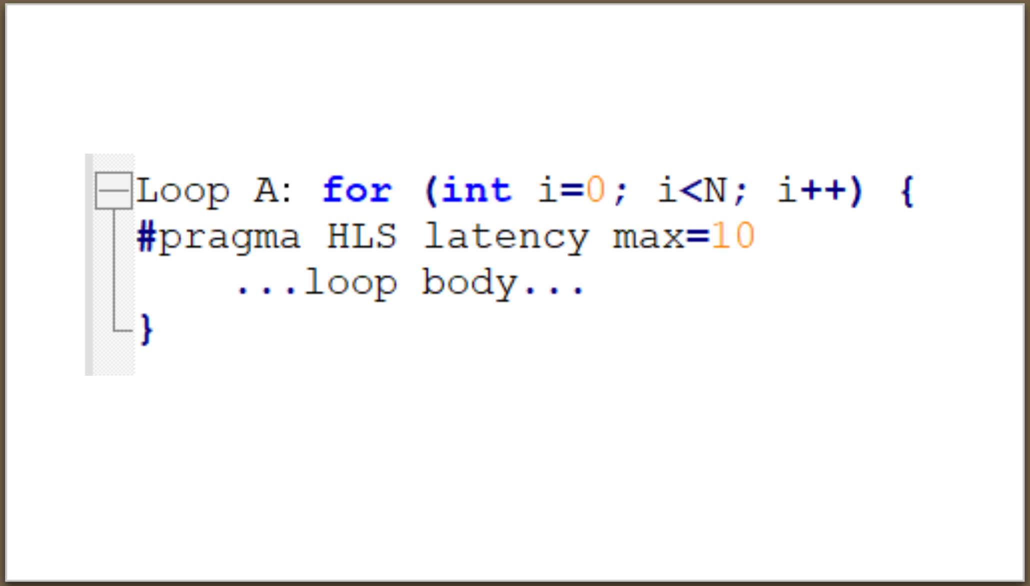
-- If the parameter is an array, Vivado HLS implements the channel as a ping-pong buffer or a FIFO as follows:

- If Vivado HLS determines the data is accessed in sequential order, Vivado HLS implements the memory channel as a FIFO channel of depth 2.
- If Vivado HLS is unable to determine that the data is accessed in sequential order or in an arbitrary manner, Vivado HLS implements the memory channel as a ping-pong buffer.

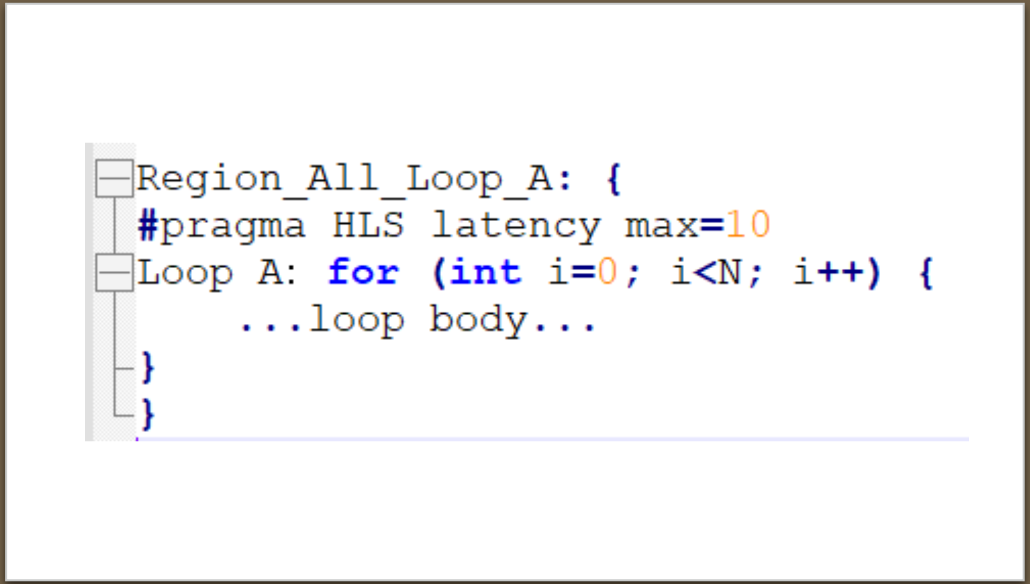
Optimizing for Latency

- Vivado HLS supports the use of a latency constraint on any scope. Latency constraints are specified using the LATENCY directive.
- When a maximum and/or minimum LATENCY constraint is placed on a scope, Vivado HLS tries to ensure all operations in the function complete within the range of clock cycles specified.

The latency directive applied to a loop specifies the required latency for a **single** iteration of the loop.



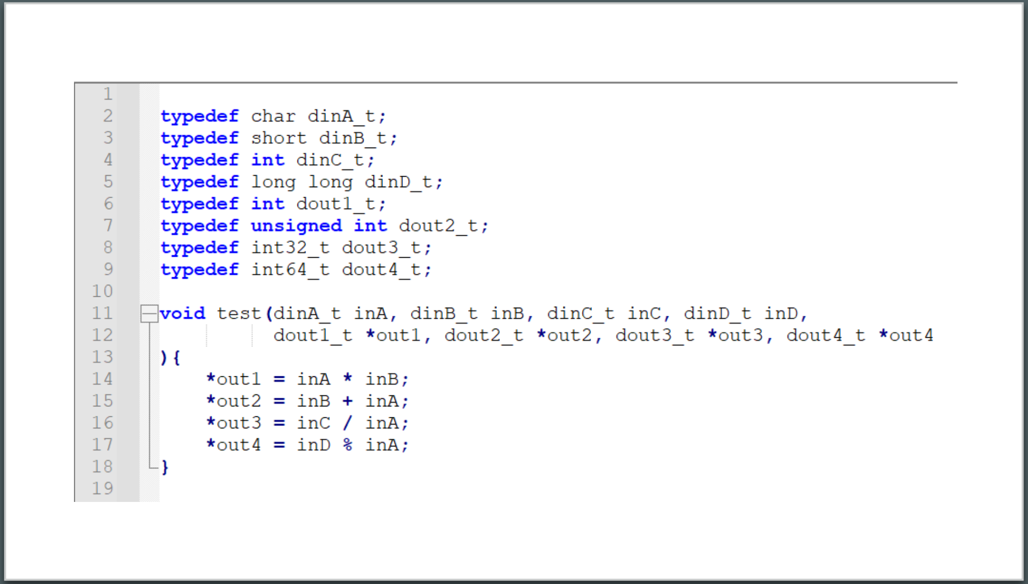
```
Loop A: for (int i=0; i<N; i++) {
#pragma HLS latency max=10
    ...loop body...
}
```



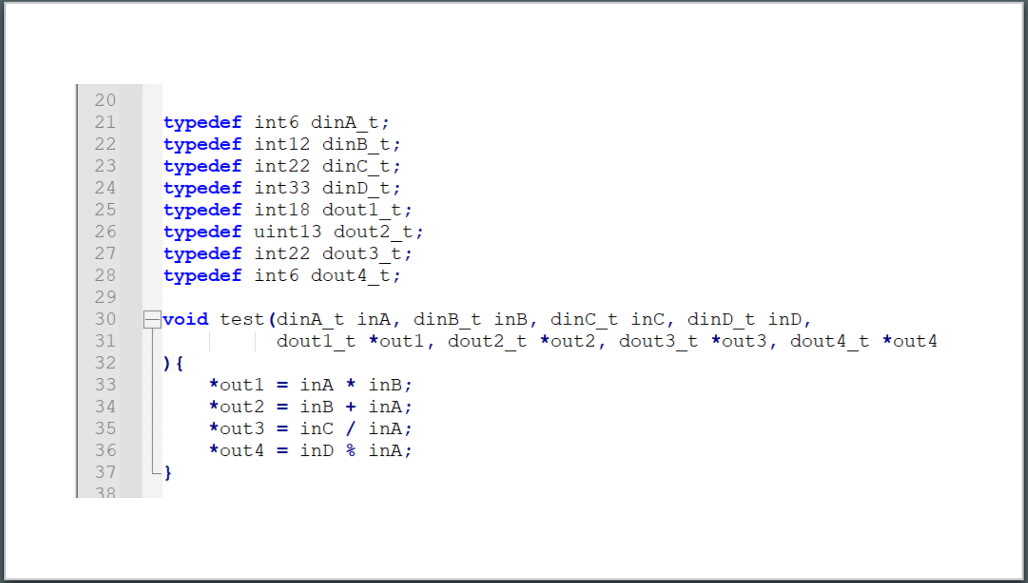
```
Region_All_Loop_A: {
#pragma HLS latency max=10
Loop A: for (int i=0; i<N; i++) {
    ...loop body...
}
}
```

Optimizing for Area

- Use the appropriate precision for the data types.
- Confirm the size of any arrays that are to be implemented as RAMs or registers. The area impact of any over-sized elements is wasteful in hardware resources.
- Pay special attention to multiplications, divisions, modulus or other complex arithmetic operations. If these variables are larger than they need to be, they negatively impact both area and performance.



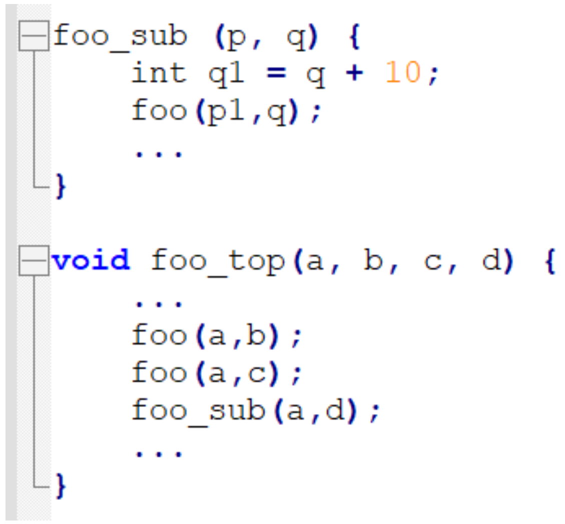
```

typedef char dinA_t;
typedef short dinB_t;
typedef int dinC_t;
typedef long long dinD_t;
typedef int dout1_t;
typedef unsigned int dout2_t;
typedef int32_t dout3_t;
typedef int64_t dout4_t;

void test(dinA_t inA, dinB_t inB, dinC_t inC, dinD_t inD,
          dout1_t *out1, dout2_t *out2, dout3_t *out3, dout4_t *out4
){
    *out1 = inA * inB;
    *out2 = inB + inA;
    *out3 = inC / inA;
    *out4 = inD % inA;
}

```



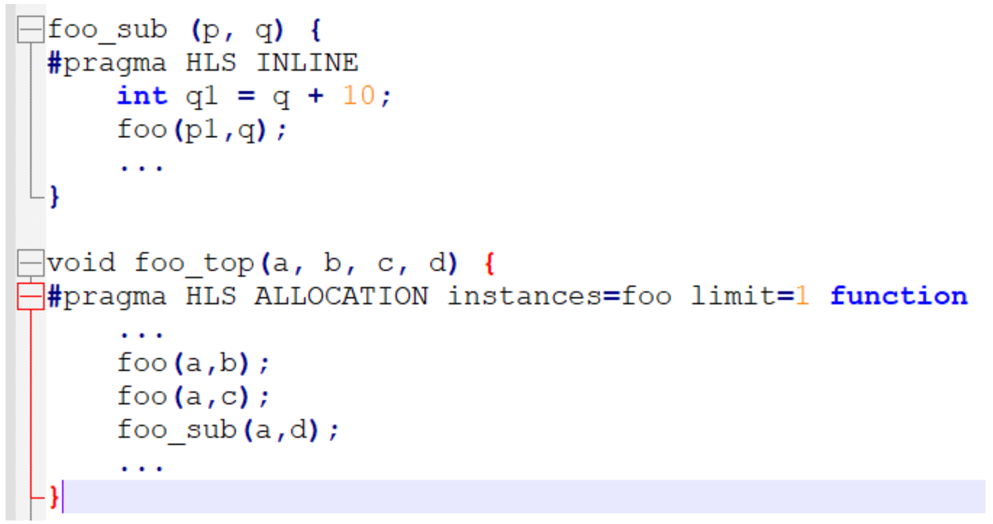
```

typedef int6 dinA_t;
typedef int12 dinB_t;
typedef int22 dinC_t;
typedef int33 dinD_t;
typedef int18 dout1_t;
typedef uint13 dout2_t;
typedef int22 dout3_t;
typedef int6 dout4_t;

void test(dinA_t inA, dinB_t inB, dinC_t inC, dinD_t inD,
          dout1_t *out1, dout2_t *out2, dout3_t *out3, dout4_t *out4
){
    *out1 = inA * inB;
    *out2 = inB + inA;
    *out3 = inC / inA;
    *out4 = inD % inA;
}

```

Optimizing for Area – Function Inlining

Function inlining removes the function hierarchy. A function is inlined using the INLINE directive. Inlining a function may improve area by allowing the components within the function to be better shared or optimized with the logic in the calling function.



```
foo_sub (p, q) {
     int q1 = q + 10;
     foo(p1,q);
     ...
}

void foo_top(a, b, c, d) {
     ...
     foo(a,b);
     foo(a,c);
     foo_sub(a,d);
     ...
}
```



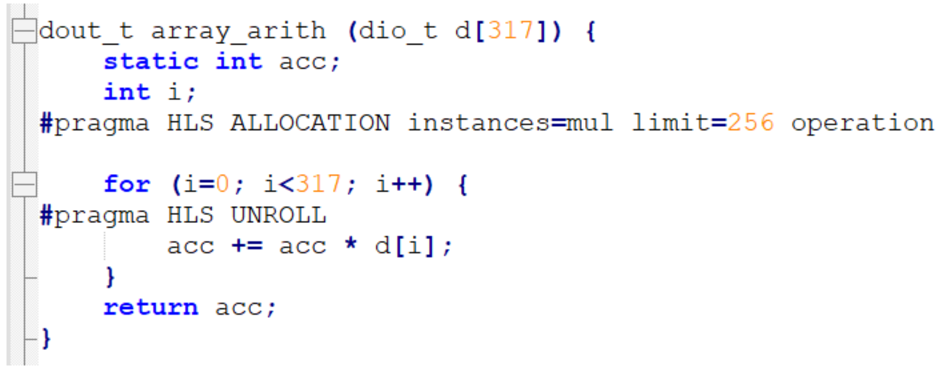
```
foo_sub (p, q) {
#pragma HLS INLINE
     int q1 = q + 10;
     foo(p1,q);
     ...
}

void foo_top(a, b, c, d) {
#pragma HLS ALLOCATION instances=foo limit=1 function
     ...
     foo(a,b);
     foo(a,c);
     foo_sub(a,d);
     ...
}
```

Optimizing for Area – Mapping Arrays into One

When there are many small arrays in the C Code, mapping them into a single larger array typically reduces the number of block RAM required.

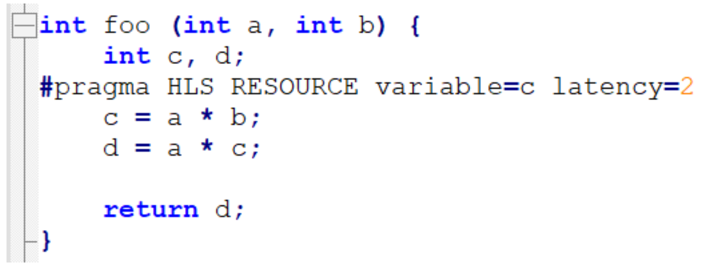
The ARRAY_MAP directive supports two ways of mapping small arrays into a larger one:
• Horizontal mapping: this corresponds to creating a new array by concatenating the original arrays. Physically, this gets implemented as a single array with more elements.
• Vertical mapping: this corresponds to creating a new array by concatenating the original words in the array. Physically, this gets implemented by a single array with a larger bit-width.

Optimizing for Area – Controlling hardware Resources



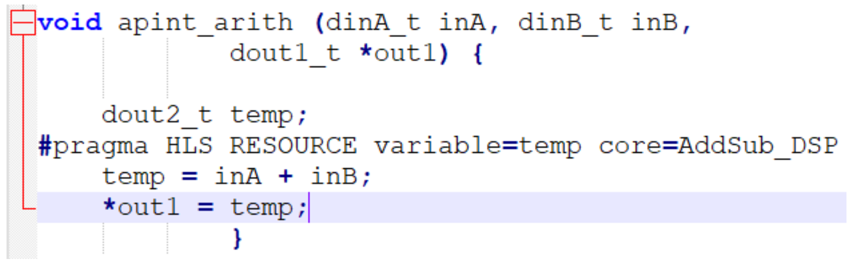
```
dout_t array_arith (dio_t d[317]) {
    static int acc;
    int i;
#pragma HLS ALLOCATION instances=mul limit=256 operation

    for (i=0; i<317; i++) {
#pragma HLS UNROLL
        acc += acc * d[i];
    }
    return acc;
}
```

If a design called foo has 317 multiplications but the FPGA only has 256 multiplier resources (DSP48s). The ALLOCATION directive shown left directs Vivado HLS to create a design with maximum of 256 multiplication operators.



```
int foo (int a, int b) {
    int c, d;
#pragma HLS RESOURCE variable=c latency=2
    c = a * b;
    d = a * c;

    return d;
}
```

The following command informs Vivado HLS to use a 2-stage pipelined multiplier for variable c. It is left to Vivado HLS which core to use for variable d.



```
void apint_arith (dinA_t inA, dinB_t inB,
        dout1_t *out1) {

    dout2_t temp;
#pragma HLS RESOURCE variable=temp core=AddSub_DSP
    temp = inA + inB;
    *out1 = temp;
        }
```

In the left example, the RESOURCE directives specify that the operation for variable temp is implemented using the AddSub_DSP core. This ensures that the operation is implemented using a DSP48 rather than LUTs which by default.

HLS vs Opencl

OPTIMIZE	XILINX OPENCL	VIVADO HLS
array_partition	__attribute__((xcl_array_partition(<type>, <factor>, <dimension>)))	#pragma HLS array_partition variable=<name> \ <type> factor=<int> dim=<int>
array_reshape	__attribute__((xcl_array_reshape(<type>, <factor>, <dimension>)))	#pragma HLS ARRAY_RESHAPE <variable> <type> <factor> <dim>
dataflow	__attribute__((xcl_dataflow))	#pragma HLS DATAFLOW
pipeline	__attribute__((xcl_pipeline_loop(II=1)))	#pragma HLS PIPELINE
unroll	__attribute__((opencl_unroll_hint(<factor>)))	#pragma HLS unroll <factor>
inline	__attribute__((always_inline))	#pragma HLS inline

HLS vs OpenCL

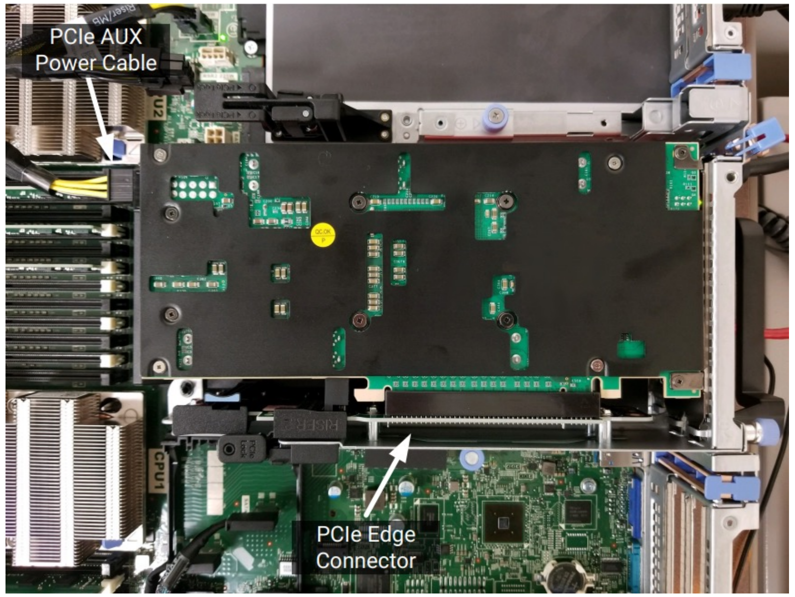
	HLS	OpenCL
Time of Exitance	30-40 years	from 2008
Scope	module， Module-level algorithm implementation	system， for heterogeneous， A complete programming model
Target Product	IP which will be integrated into system	A complete design
The user	FPGA developer， develop algorithm quickly	System developer, little impact on the circuit

OpenCL Support

-- A software programming model for software engineers and a software methodology for system architects.

-- First industry standard for heterogeneous computing.

-- Provides increased performance with hardware acceleration.



OpenCL Support

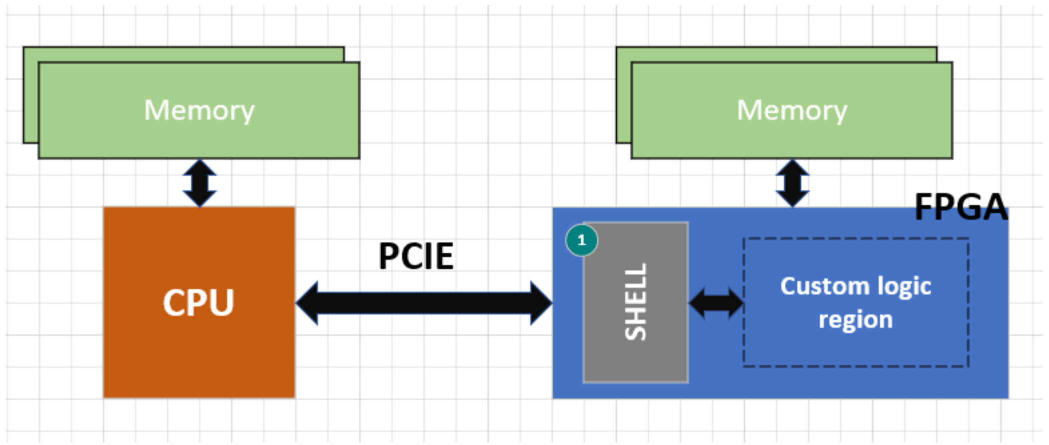
Xilinx suggests using HLS to develop FPGA kernel and using OpenCL supported software to generate host code.
Steps for application execution on a heterogeneous system are shown as below:

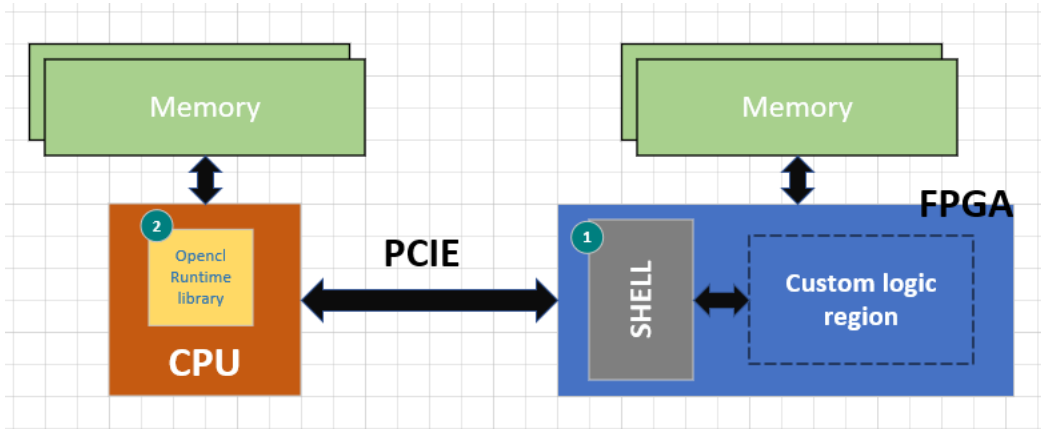
-- Powering up
-- Initializing runtime components
-- Configuring the device
-- Allocating buffers
-- Writing the buffers to FPGA memory
-- Running the accelerators
-- Reading the buffers from FPGA memory

OpenCL Support

1. On power up, only the shell is initialized in the FPGA. The shell will manage communication with the host.

2. Devices = get_devices(“Xilinx/Altera”);
   cl::Device device = devices[0];
   cl::Context context(…);
   cl::CommandQueue q(context, device, …);

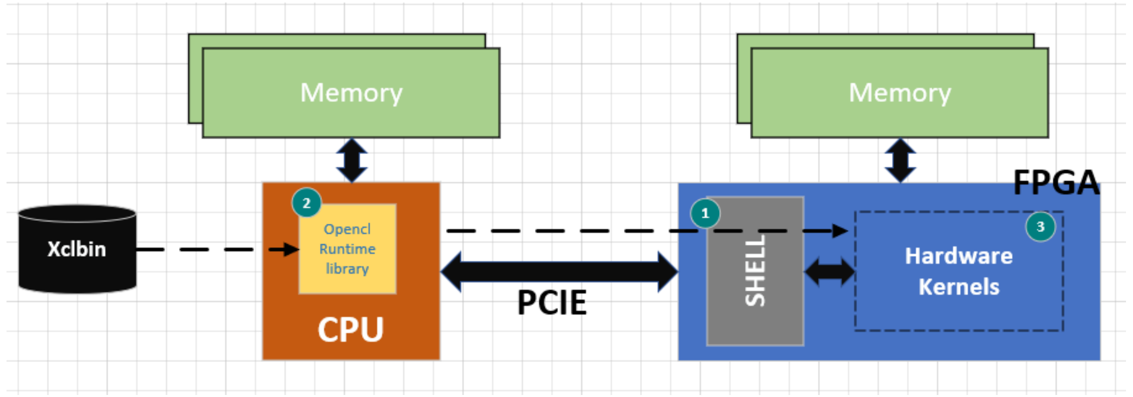


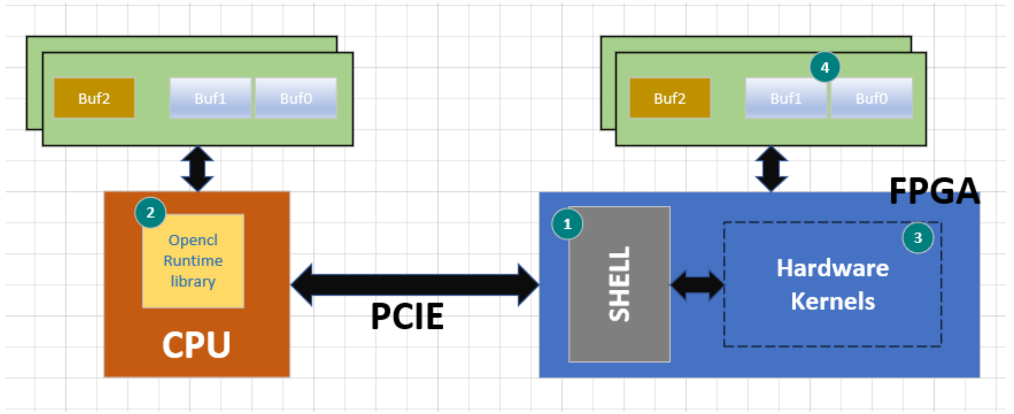


OpenCL Support

3. cl::Program::Binaries bins{{fileBuf,
   FileBufSize}};
   cl::Program program(context, devices
   bins, …);

4. buf0 = cl::clCreateBuffer(context, CL_MEM_READ_ONLY,…);
   buf1 = cl::clCreateBuffer(context, CL_MEM_READ_ONLY, …);
   buf2 = cl::clCreateBuffer(context, CL_MEM_WRITE_ONLY,…);

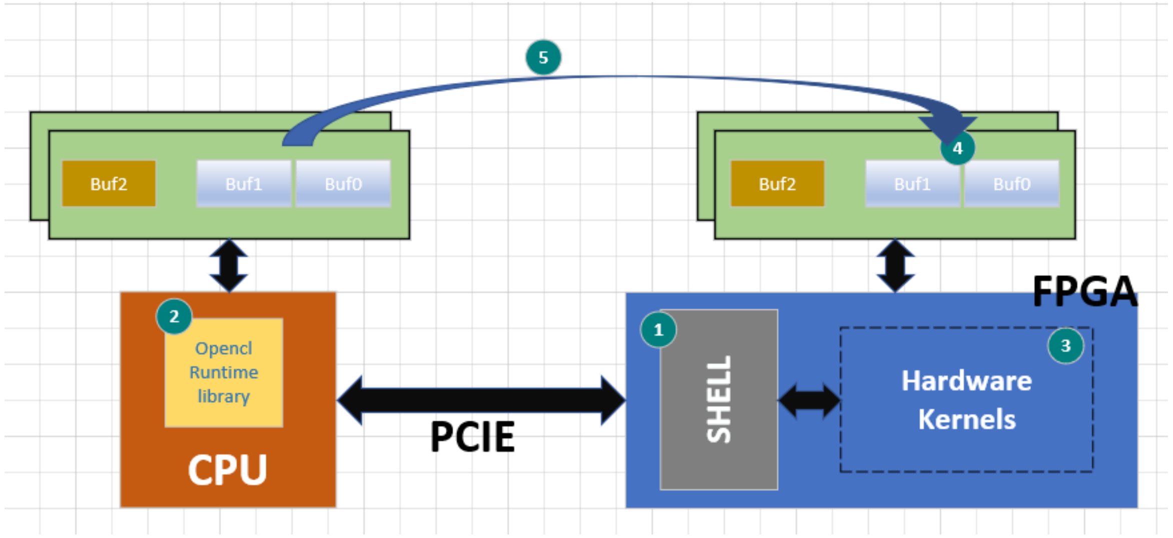


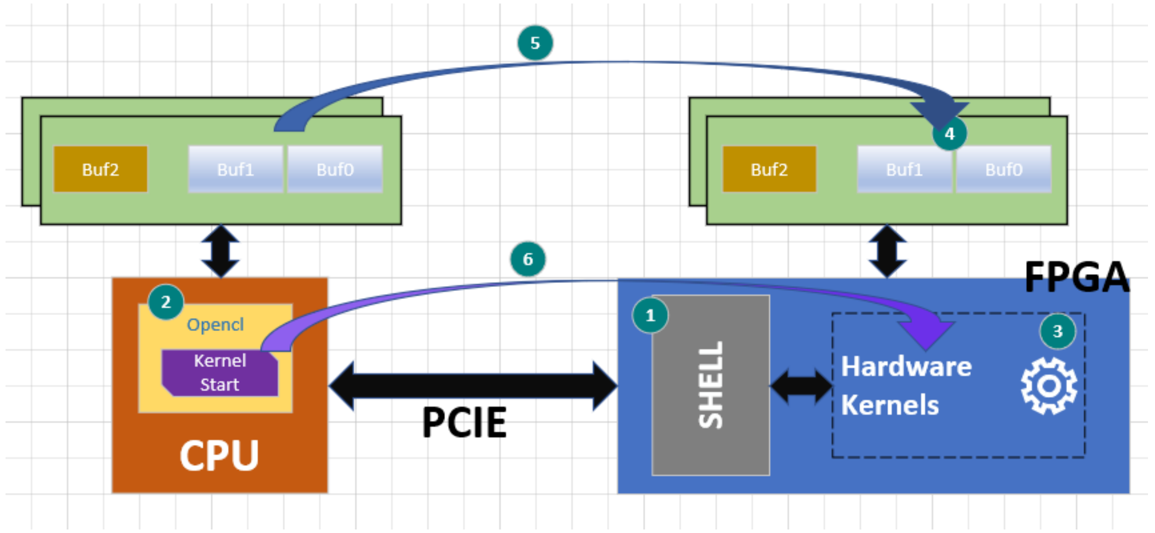


OpenCL Support

5. clEnqueueMigrateMemObjects (Command_Queue, 1, &GlobMem_BUF_DataIn_1, …);

6. Kernel = cl::clCreatekernel(program, “mykernel”, …);
   cl::clSetKernelArg(kernel, 0, sizeof(cl_mem), &buf0);
   cl::clSetKernelArg(kernel, 1, sizeof(cl_mem), &buf1);
   cl::clEnqueueTask(Command_Queue, Kernel, 0, NULL, NULL);



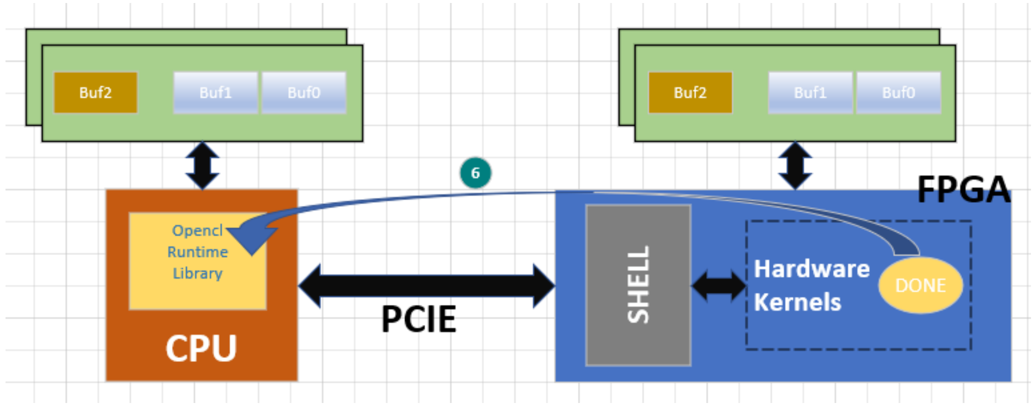


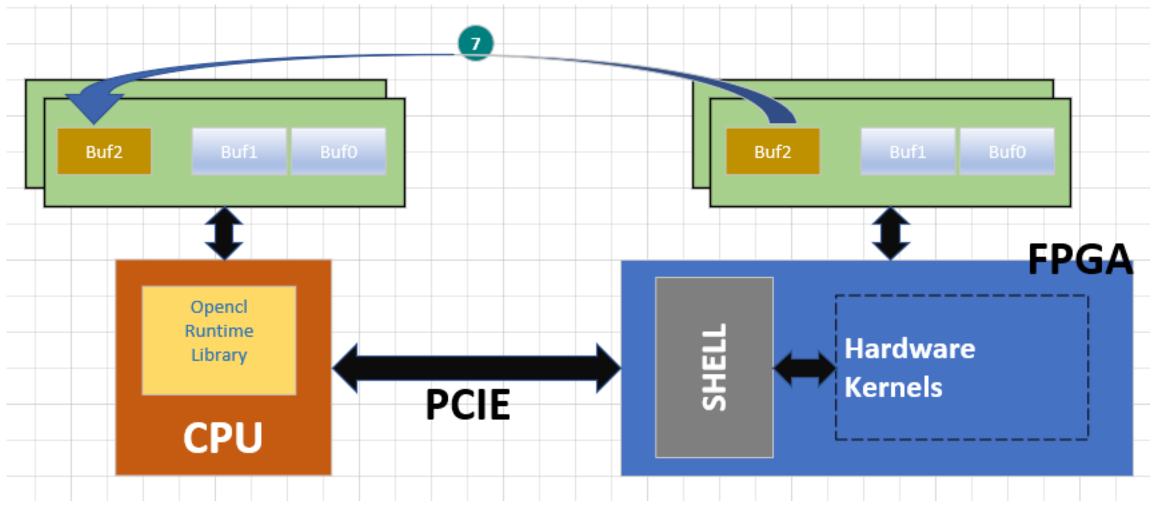
OpenCL Support

6(cont'd).

cl::clFinish(Command_Queue);

7. cl::clEnqueueMigrateMemObjects(Command_Queue, 1, &GlobMem_BUF_RES, CL_MIGRATE_MEM_OBJECT_HOST,…);





The End