Vivado HLS

SDx 2018.2

XILINX.

Objectives

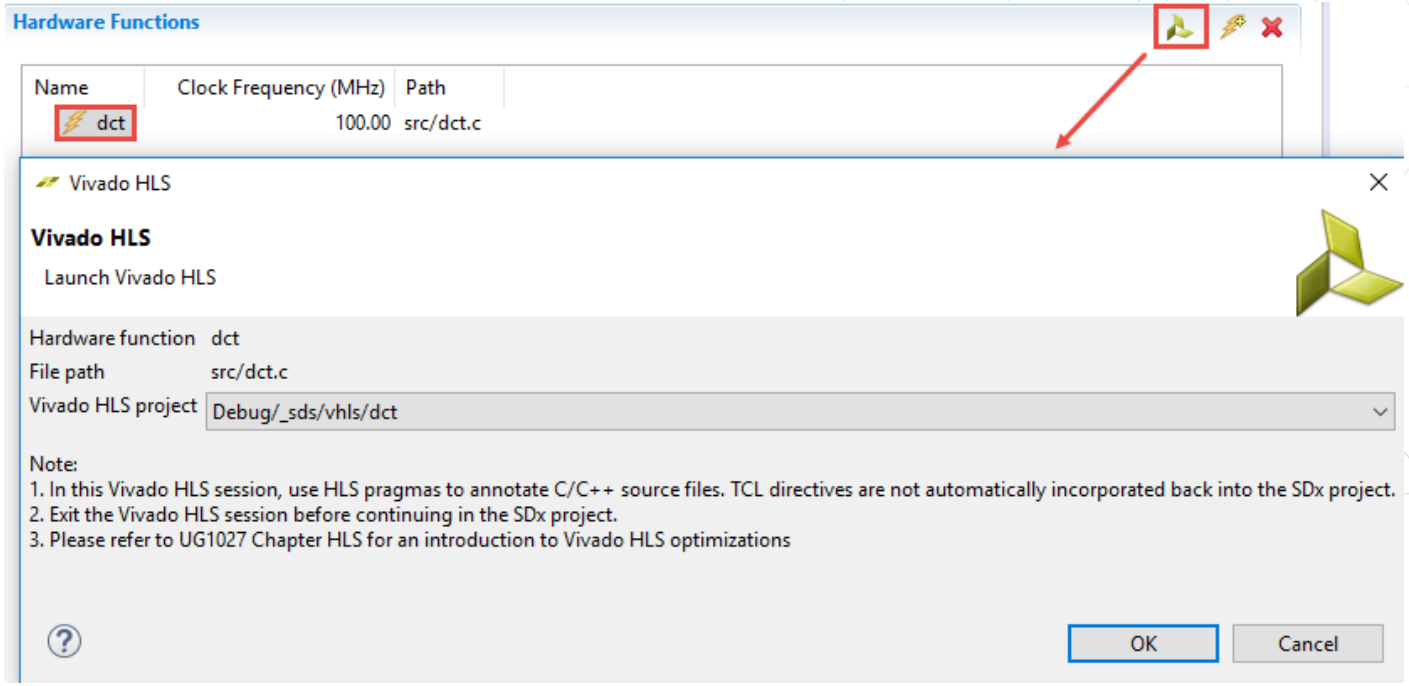
- **After completing this module, you will be able to:**
  - Describe HLS use models in SDSoC
  - List function arguments limitations
  - Explain various optimization techniques
  - Describe what dataflow is

Outline

- HLS Use Models
- Optimizing Techniques
- HLS Libraries
- Summary
- Lab6 Intro

Relevant HLS Use Models

- **Vivado HLS as IP development environment**
  - Export IP from HLS and import as C-callable IP
- **Vivado HLS as Zynq PL cross-compiler**
  - Hardware function source code shared between SDSoC and VHLS and ***the only*** handoff files
  - VHLS projects are automatically created by SDSoC and are temporary work products (no persistent state)
  - Optionally launch HLS from SDSoC
    - Optimize accelerator code
    - Simulate hardware function



XILINX.

Hardware-aware Software Programming

The most common and important HLS compiler directives are natural to performance-oriented software programmers

- **Use hardware buffers to improve communication between accelerator and external memory**
  - Copy loops at the function boundary when multiple accesses required and to burst data into local buffers
- **Inline functions to improve boundary optimization**
- **Pipeline and unroll inner loops to increase concurrency**
- **Partition arrays to increase memory bandwidth**
- **Use “dataflow” for concurrent execution of independent blocks of code within an HLS function**
  - SDSoC automatically implements dataflow pipelining between cascaded hardware functions

Hardware Function Argument Interfaces

- **Use standard C99 type arguments at the top-level interface**
  - Avoid `long` and `bool B[16]`
  - Avoid arrays of struct (in future, SDSoC will support packed, gcc-compatible arrays)
  - Avoid `ap_int<>`, `ap_fixed<>`, `hls::stream` except widths `8:16:32:64`
    - Must use `#ifndef __SDS_VHLS__` to coerce to like-sized C99 type
      - `sdscc/sds++` always passes `__SDS_VHLS__` macro to HLS
    - `hls::stream` arguments must be presented to `sdscc/sds++` as arrays
- **Let SDSoC generate HLS interface directives**
  - Use `#pragma SDS data access_pattern(a:RANDOM, b:SEQUENTIAL)`
    - Employ `#pragma HLS interface` only to override
  - Pitfall: incomplete HLS interface pragma specification can result in cryptic errors
- **Pitfall: "simulates in HLS" does not imply "correct in SDSoC"**
  - Examples: missing `#pragma HLS interface s_axilite port=return`
    - `#pragma HLS interface s_axilite port=foo`
    - `#pragma HLS interface m_axi port=foo offset=slave`

Optimizing Techniques

Optimization Methodology

Baseline Hardware Function	• Determine performance with compiler defaults
1: Optimization for Metrics	• Define loop trip counts
2: Pipeline for Performance	• Pipeline and Dataflow
3: Optimize Structures for Performance	• Partition memories • Remove false dependencies
4: Reduce Latency	• Optionally specify latency requirements
5: Improve Area	• Optionally recover resources through sharing

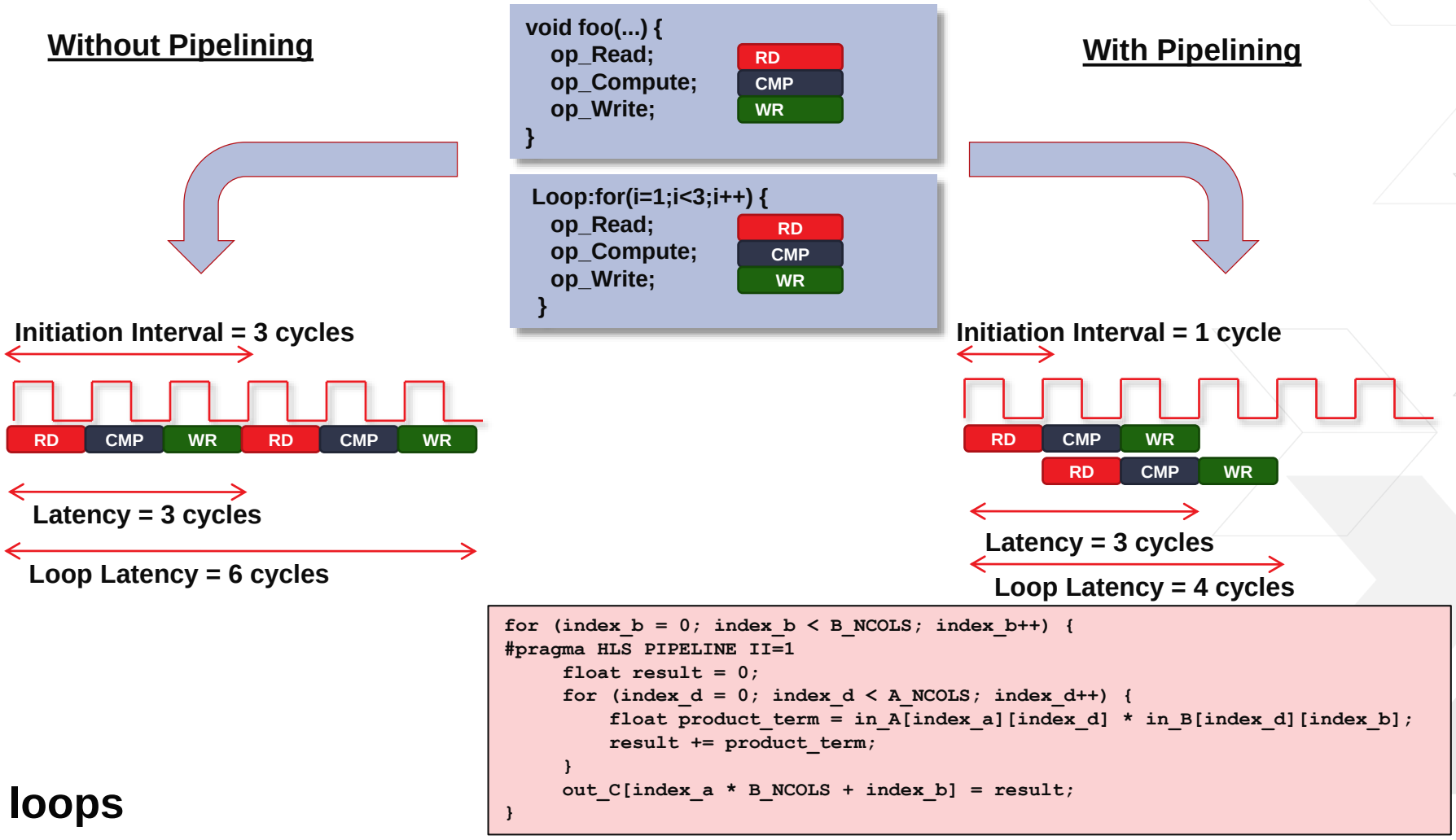
Primary Optimization Techniques

- **Memory access can be the biggest performance bottleneck**
  - May have to restructure algorithm to optimize memory accesses for hardware (similar issues for GPU, DSPs)
  - Pointers are an efficient way to pass ownership to software functions, but not necessarily to hardware functions
  - Burst data into local arrays using memcpy or copy loops (e.g., line buffer, small matrices)
- **Increase local concurrency**
  - Pipeline and dataflow loops, function calls, and operations
  - Enhance pipeline performance

HLS Optimization Pragmas

Directives and Configurations	Description
PIPELINE	Reduces the initiation interval by allowing the concurrent execution of operations within a loop or function
DATAFLOW	Enables task level pipelining, allowing functions and loops to execute concurrently. Used to minimize interval
INLINE	Inlines a function, removing all function hierarchy. Used to enable logic optimization across function boundaries and improve latency/interval by reducing function call overhead
UNROLL	Unroll for-loops to create multiple independent operations rather than a single collection of operations
ARRAY_PARTITION	Partitions large arrays into multiple smaller arrays or into individual registers, to improve access to data and remove block RAM bottlenecks

Loop and function pipelining

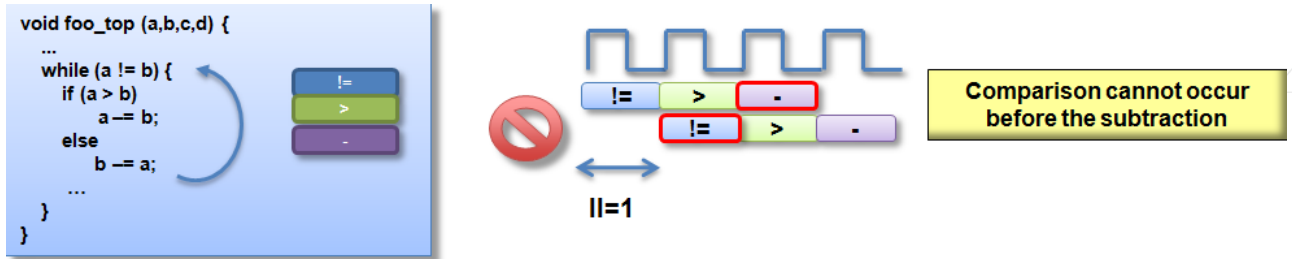



```
for (index_b = 0; index_b < B_NCOLS; index_b++) {
#pragma HLS PIPELINE II=1
     float result = 0;
     for (index_d = 0; index_d < A_NCOLS; index_d++) {
         float product_term = in_A[index_a][index_d] * in_B[index_d][index_b];
         result += product_term;
     }
     out_C[index_a * B_NCOLS + index_b] = result;
}
```

- **Pipelined loops**
  - Combined with array partitioning to achieve II=1

Issues Which Prevent Pipelining

- **Pipelining functions unrolls all loops**
  - Loops with variable bounds cannot be unrolled
  - This will prevent pipelining
    - Re-code to remove the variables bounds: max bounds with an exit
- **Feedback prevent/limits pipelines**
  - Feedback within the code will prevent or limit pipelining
    - The pipeline may be limited to higher initiation interval (more cycles, lower throughput)

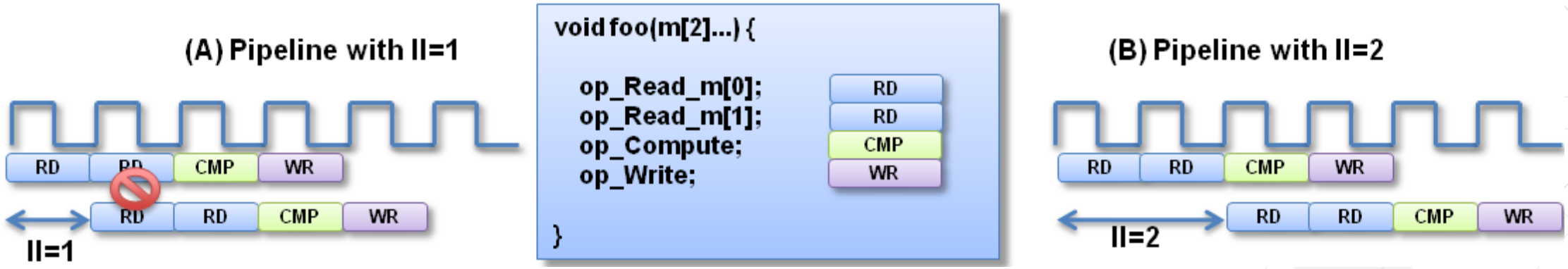


- **Resource Contention may prevent pipelining**
  - Can occur within input and output ports/arguments
  - This is a classis way in which arrays limit performance

XILINX.

Resource Contention: Unfeasible Initiation Intervals

- **Sometimes the II specification cannot be met**
  - In this example there are 2 read operations on the same port



  - An II=1 cannot be implemented
    - The same port cannot be read at the same time
    - Similar effect with other resource limitations
    - For example if functions or multipliers etc. are limited
- **Vivado HLS will automatically increase the II**
  - Vivado HLS will always try to create a design, even if constraints must be violated

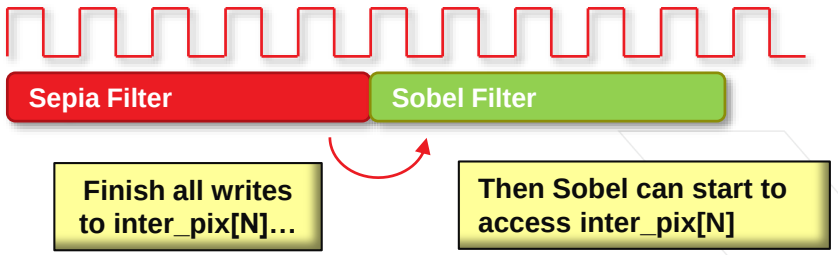
Dataflow

- **Default behavior**
  - Complete a function or loop iteration before starting next function or loop iteration

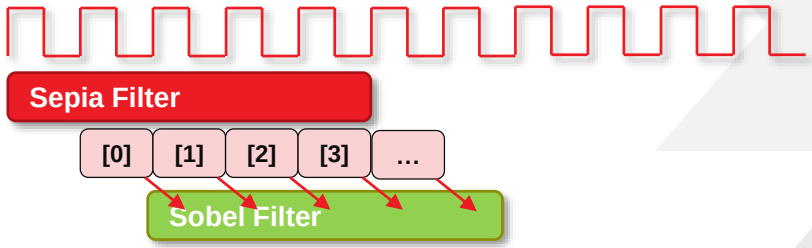
```
//This memory is turned into a FIFO during optimization
 rgb_pixel inter_pix[MAX_HEIGHT][MAX_WIDTH];

// Primary processing functions
 sepia_filter(in_pix,inter_pix);
 sobel_filter(inter_pix,out_pix2);
```

Sepia Filter
Sobel Filter



- **Dataflow**
  - Start next function or loop iteration as soon as “ready” and data is available
  - Initiation interval (II) represents number of clocks between ‘starts’
  - Increased concurrency
  - Buffers data between processes
    - Worst case 2-BRAM (ping-pong)
    - Optimized case, 1 reg (1 element FIFO)

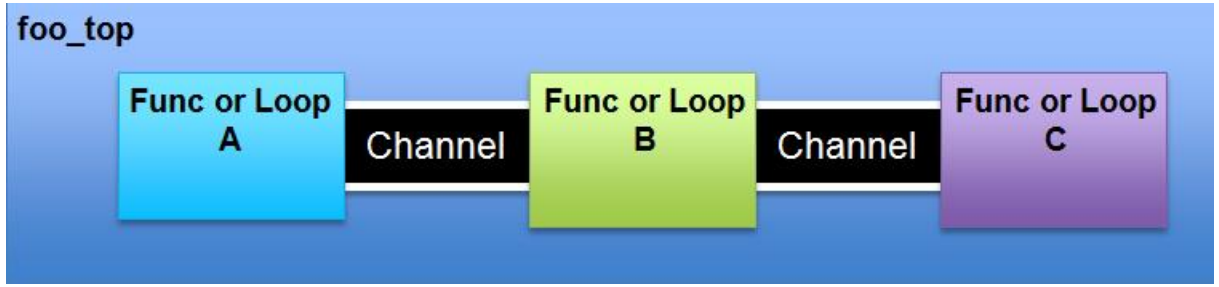


- **Apply dataflow within a function but not at the top level**

XILINX.

Dataflow Optimization

- **Dataflow Optimization**
  - Allows blocks of code to operate concurrently
    - The blocks can be functions or loops
    - Dataflow allows loops to operate concurrently
  - It places channels between the blocks to maintain the data rate



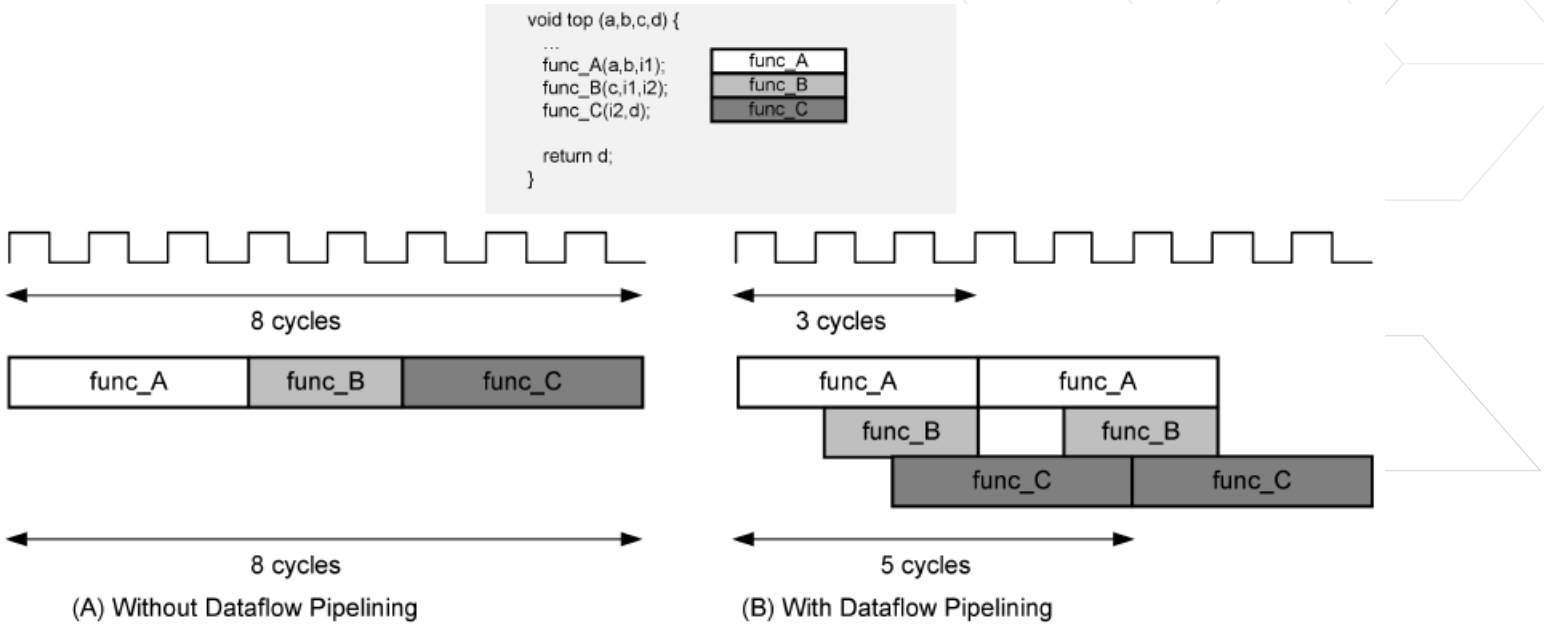
    - For arrays the channels will include memory elements to buffer the samples
    - For scalars the channel is a register with hand-shakes

- **Dataflow optimization therefore has an area overhead**
  - Additional memory blocks are added to the design
  - The timing diagram on the previous page should have a memory access delay between the blocks
    - Not shown to keep explanation of the principle clear

Dataflow Pipelining

- **Function dataflow pipelining**
  - Use `#pragma HLS dataflow` where the data flow optimization is desired
  - Example:

```
void top(a, b, c, d) {
#pragma HLS dataflow
func_A(a, b, i1);
func_B(c, i1, i2);
func_C(i2, d);
}
```



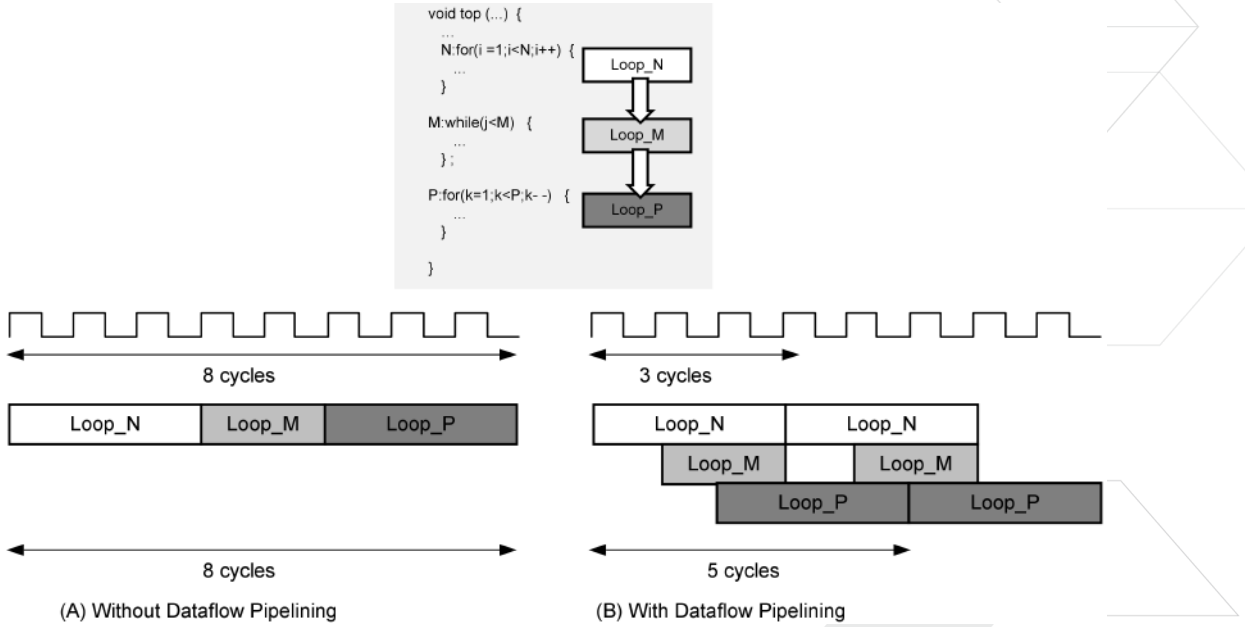
XILINX.

Dataflow Pipelining (2)

- **Loop dataflow pipelining**
  - Enables a sequence of loops, normally executed sequentially, to execute concurrently
- Use `#pragma HLS dataflow` where the data flow optimization is desired
  - Example:

```
void top(a, b, c, d) {
#pragma HLS dataflow
N: for(;;) { }
M: for(;;) { }
P: for(;;) { }
}
```

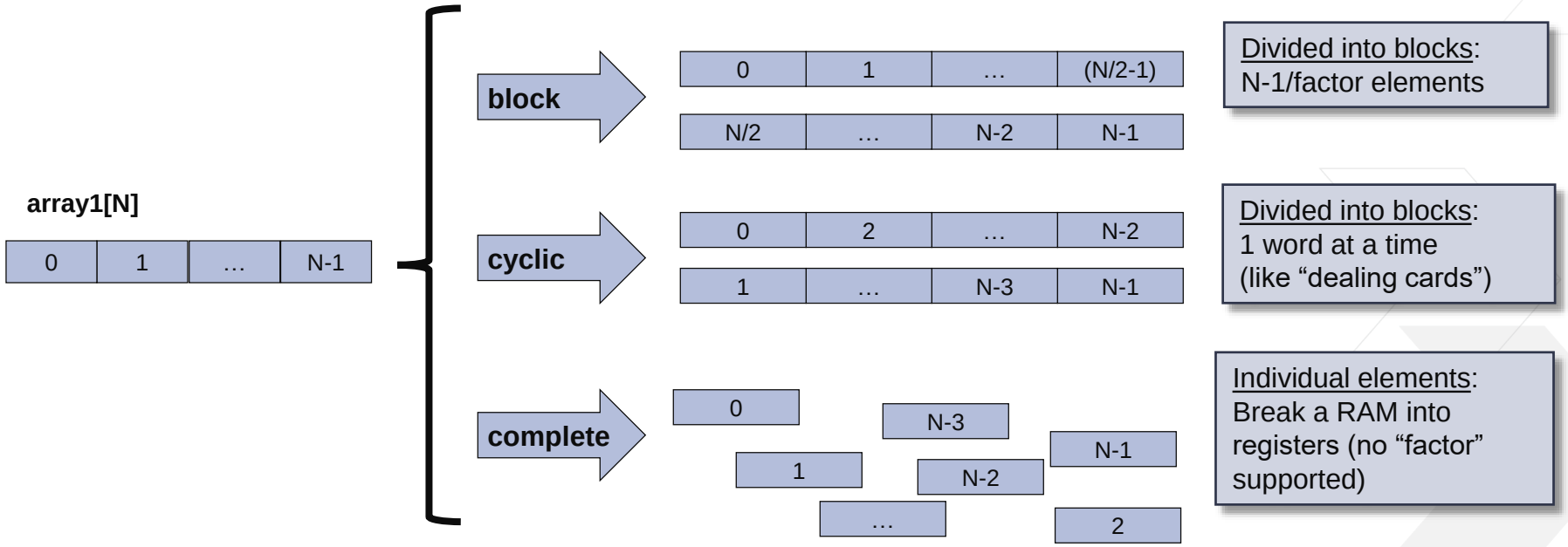
- Do not apply on a scope which contains a mixture of loops and functions



XILINX.

Array Partitioning

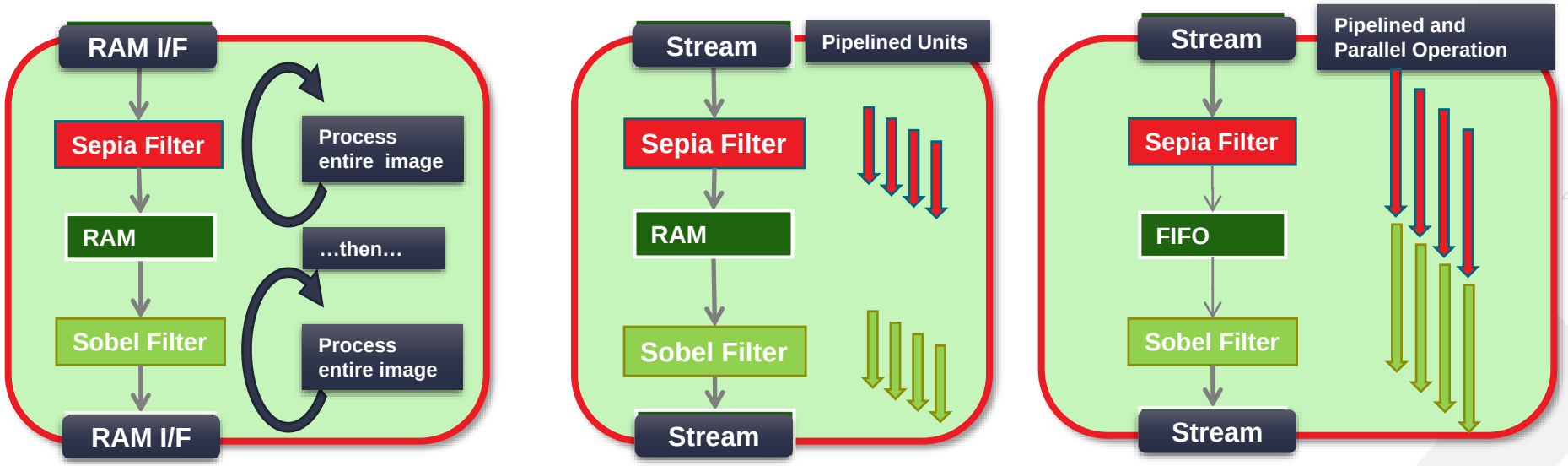
> **Partition into multiple memories to increase concurrent access**




```
void mmult_kernel(float in_A[A_NROWS][A_NCOLS], float in_B[A_NCOLS][B_NCOLS], float out_C[A_NROWS*B_NCOLS])
{
#pragma HLS INLINE self
#pragma HLS array_partition variable=in_A block factor=16 dim=2
#pragma HLS array_partition variable=in_B block factor=16 dim=1
// snip
}
```

Pipelining and Dataflow

- **Optimizations for cascaded loops, function calls, and operators**
  - Eliminate or reduce dependencies across pipelines



	Baseline	*Pipelined*	*Dataflow*
BRAM	2792	2790	**24**
FF	**891**	**1136**	**883**
LUT	**2315**	**2114**	**1606**
Interval	128,744,588	4,150,224	**2,076,613**

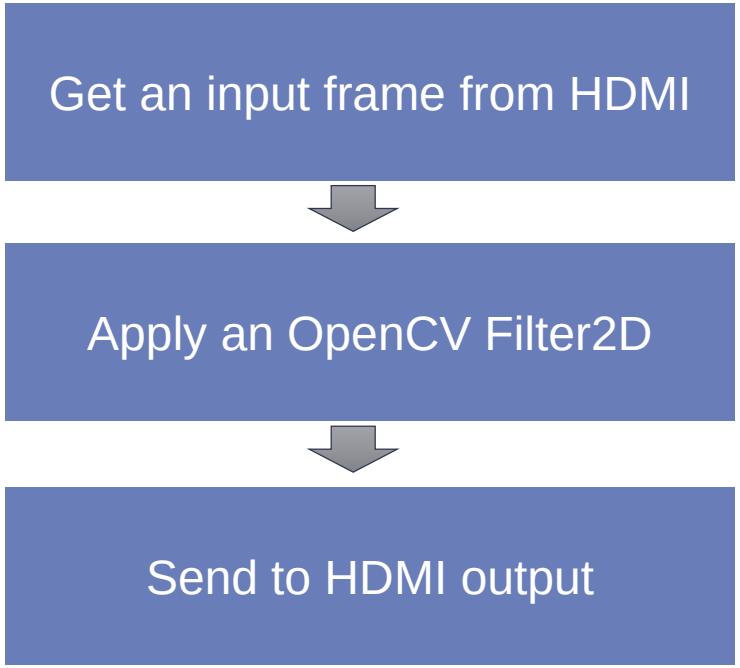
HLS Libraries

HLS Libraries

- Vivado HLS libraries are provided in the SDSoC environment
- Ensure that the source code conforms to the following rules
  - Hardware function arguments resolve to a C99 basic arithmetic type, a pointer to, array of, or a struct whose members flatten to a C99 basic arithmetic type
  - Scalar arguments must fit in a 32-bit container
  - Provide a C/C++ wrapper function to ensure the functions export a software interface

```
#ifndef _MY_SQRT_H_
#define _MY_SQRT_H_
#ifdef __SDSVHLS__
#include "hls_math.h"
#else
// The hls_math.h file includes hdl_fpo.h which contains actual code and
// will cause linker error in the ARM compiler, hence we add the function
// prototypes here
static float sqrtf(float x);
#endif
void my_sqrt(float x, float *ret);
#endif // _SQRT_H_
```

Basic Filter2D OpenCV Application




```
#include "image_filters.h"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
using namespace cv;

void image_filters( int array_in[1080*2048], int array_out[1080*2048])
{

      const char coef[3][3] =  {{-1,-2,-1},
                                { 0, 0, 0},
                                { 1, 2, 1}};

      Mat src(1080, 2048, CV_8UC4, (int *)array_in);
      Mat dst(1080, 2048, CV_8UC4, (int *)array_out);
      Mat kernel(3, 3, CV_8SC1, (char *)coef);
      Point anchor;
      double delta = 0;
      int ddepth = -1;
      anchor = Point( -1, -1 );
      filter2D(src, dst, ddepth , kernel, anchor, delta, BORDER_DEFAULT );
}
```

Summary

Summary

- **Vivado HLS can be used as an IP development environment to create C-callable IP**
- **Vivado HLS is used as a Zynq PL cross-compiler in SDSoC**
- **Function arguments of the top-level hardware functions must resolve to a C99 basic arithmetic type, a pointer to, array of, or a struct whose members flatten to a C99 basic arithmetic type**
- **When needed the hardware function performance optimization can be achieved by using pipelining, dataflow, unrolling, and array partitioning**
- **Dataflow optimization places channels between the blocks to maintain the data rate**
  - The channels could be a ping-pong buffer or a FIFO
    - The ping-pong buffer is used when the data are accessed in a random order
    - The FIFO is used when the data are accessed sequentially

Lab6 Intro

Lab6 Intro

- **Introduction**
  - This lab introduces various techniques and directives of Vivado HLS which can be used in SDSoC to improve design performance. The design under consideration performs discrete cosine transformation (DCT) on an 8x8 block of data
- **Objectives**
  - Improve performance using PIPELINE directive
  - Understand DATAFLOW directive functionality
  - Apply memory partitions technique to improve data access

Adaptable.
Intelligent.

XILINX