



Vivado HLS – Tips and Tricks

Presented By

Frédéric Rivoallon

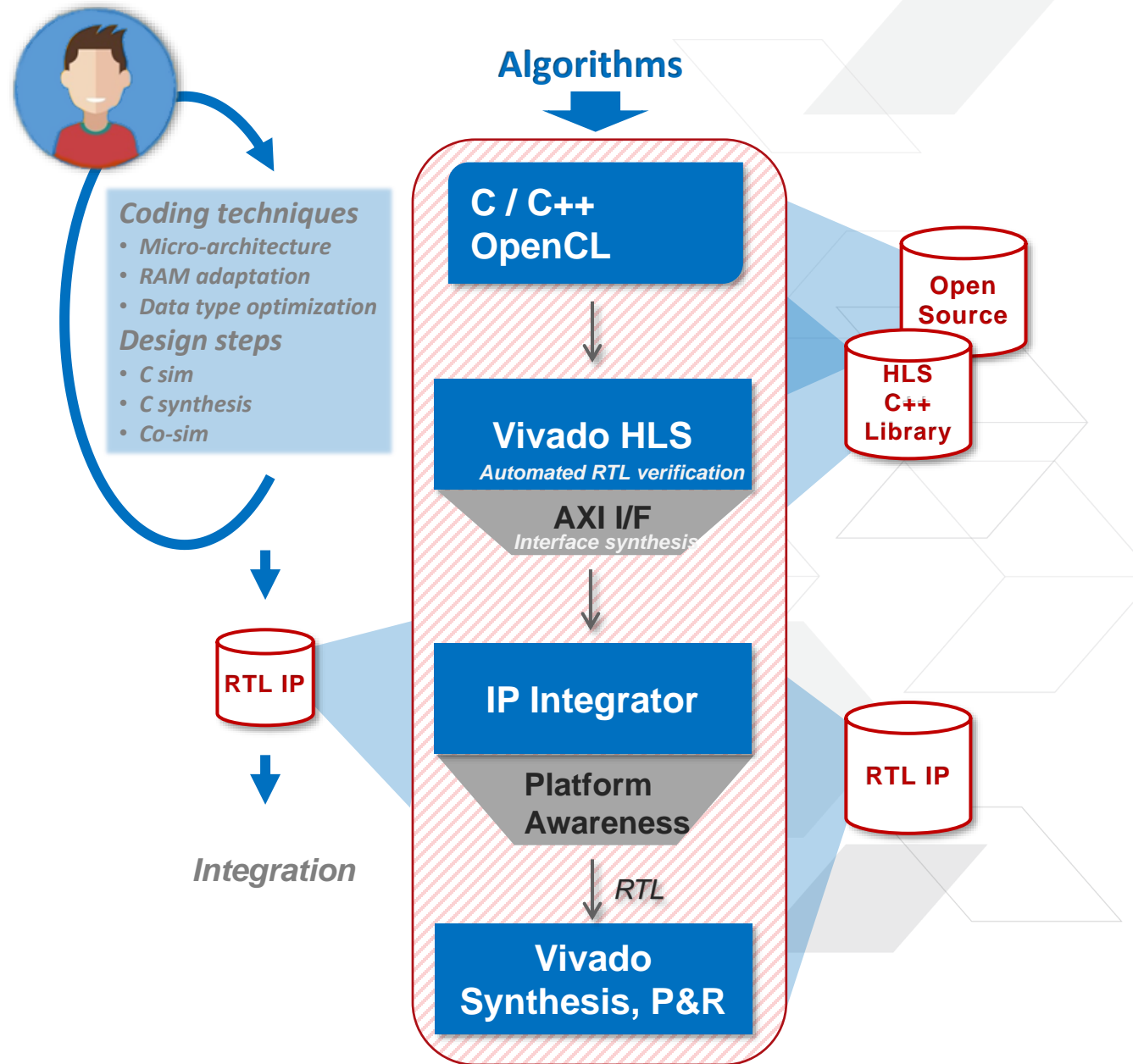
Marketing Product Manager

October 2018

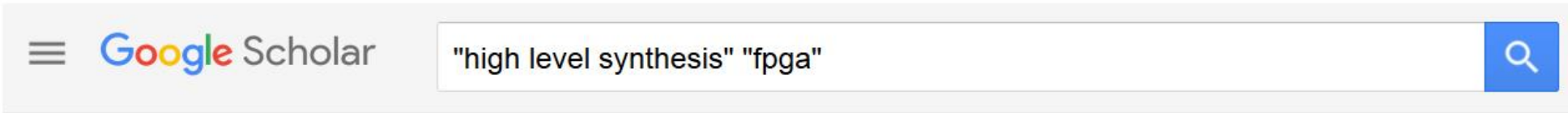


Vivado HLS

- > Abstracted C based descriptions
- > Higher productivity
 - >> Concise code
 - >> Optimized libraries
 - >> Fast C simulation
 - >> Automated simulation of generated RTL
 - >> Interface synthesis (AXI-4)



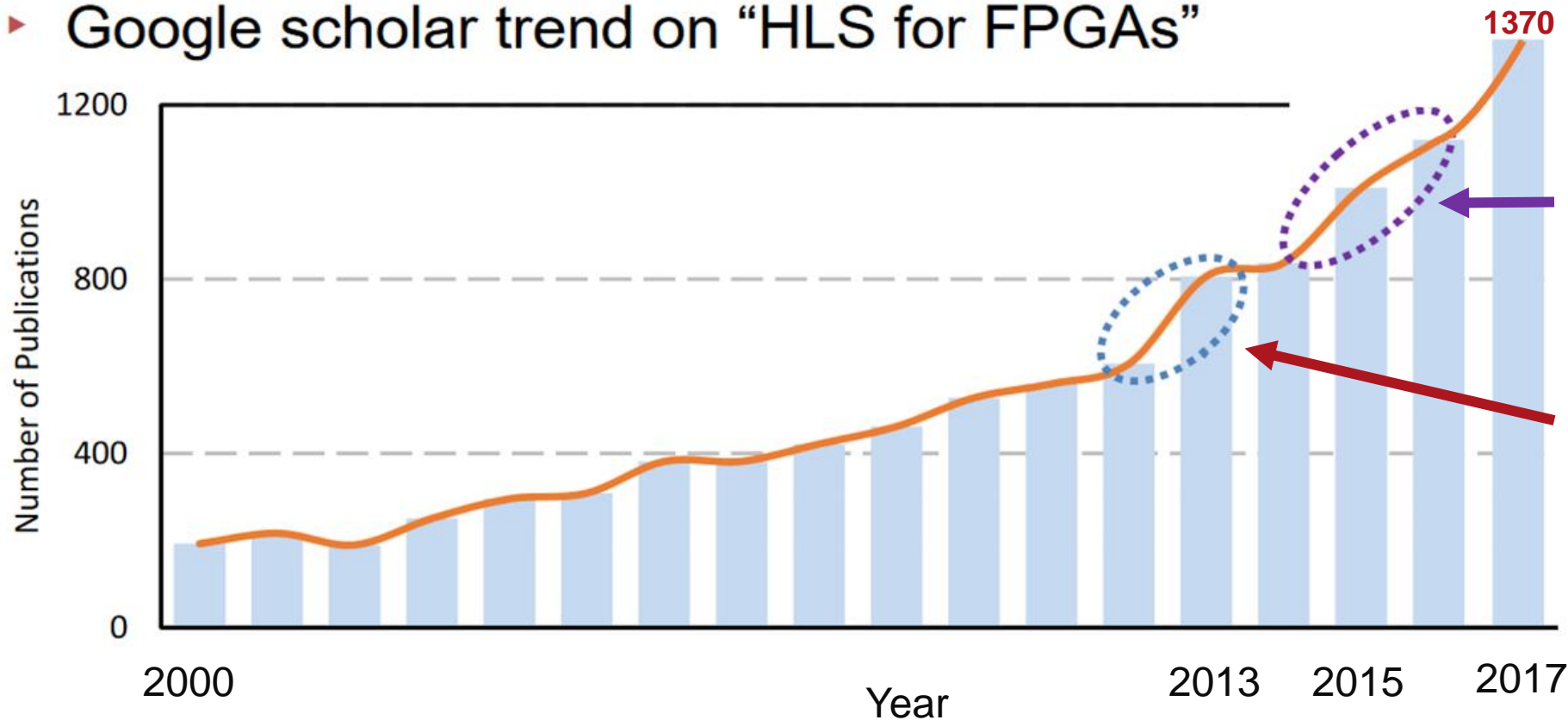
Vivado HLS Acceptance Grows...



About 5,180 results (0.05 sec)

5,000+ papers since 2014!

Google scholar trend on "HLS for FPGAs"



High demand for deep learning accelerators on FPGAs

Software programmable FPGA SoCs become available

Based on graph from Cornell University.

Factors for Overall System Performance

> Platform

Fixed Performance...

- >> Off-chip memory, data links (e.g. PCIe)
- >> Connectivity IPs

> Compute Customization

- >> Micro-architecture, parallelism, operators

> Memory Adaptation

- >> On-chip memory, shift registers, piping

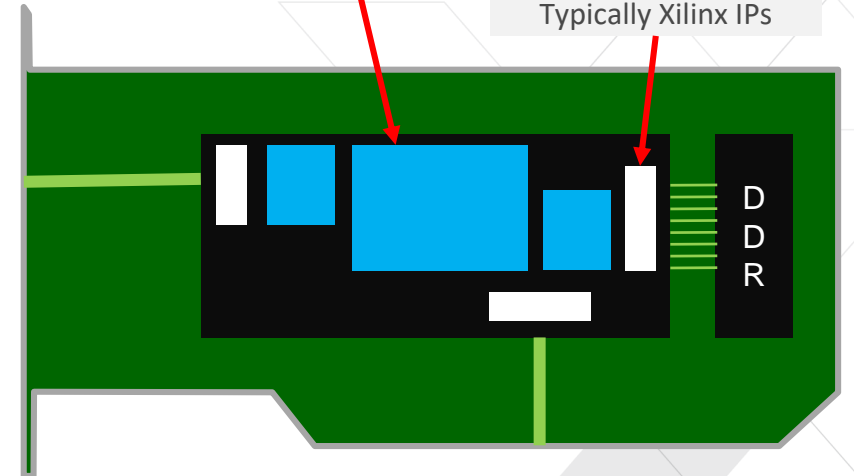
> Datatype Optimization

- >> Customized data type (adjusted to requirement)

Malleable Performance...

Data Processing (RTL, HLS)

Connectivity IPs
Typically Xilinx IPs



Identify the Performance Challenge

- > Compute-bound or memory-bound?
- > What kind of parallelism is required?

Table 1: The current set of the Rosetta applications – Rosetta contains both compute-bound and memory-bound applications with different workloads. Kernels in each application expose different sources of parallelism: SLP = subword-level parallelism; DLP = data-level parallelism; **ILP = instruction-level parallelism**. Different types of parallelism available in each compute kernel are listed in parentheses.

Algorithm Examples

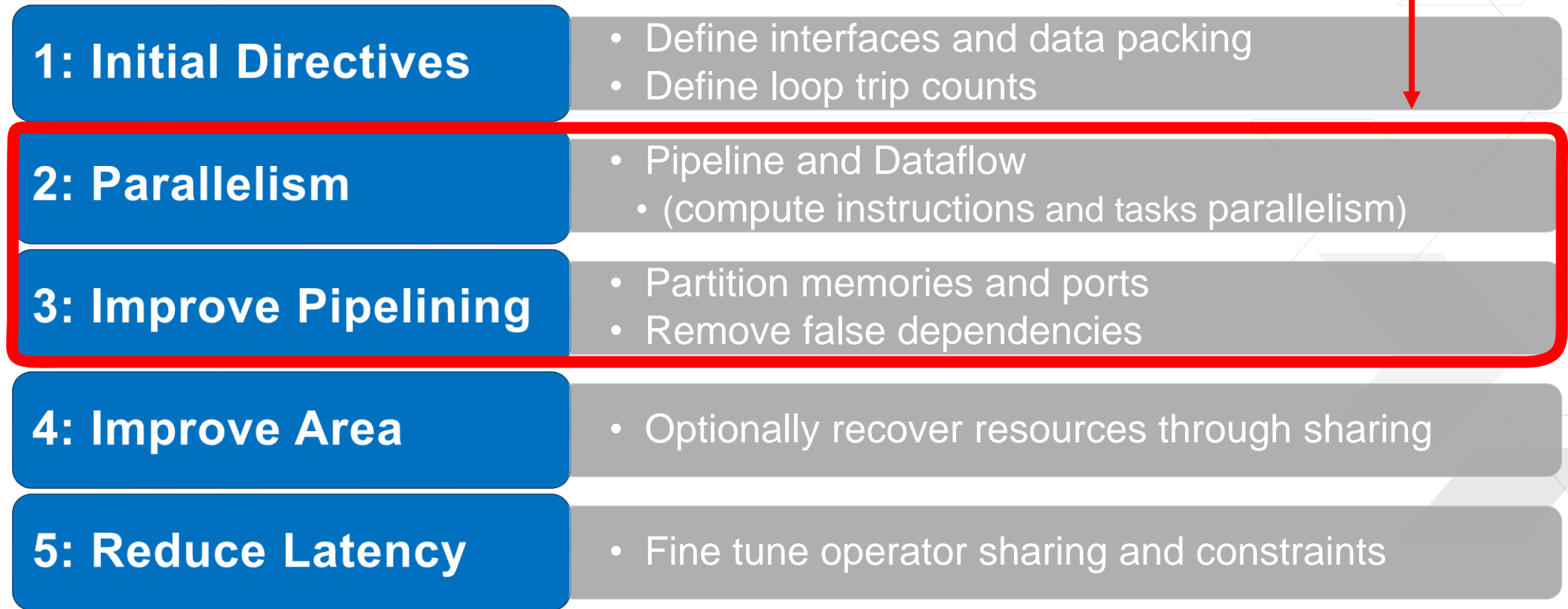
Application	Categorization	Major Compute Kernels	Major HLS Optimizations
3D Rendering	Video processing Compute bound Integer operation intensive	Integer arithmetics (ILP)	Dataflow pipelining Communication customization
Digit Recognition	Machine learning Compute bound Bitwise operation intensive	Hamming distance (SLP, DLP, ILP) KNN voting (ILP)	Loop unrolling Loop pipelining
Spam Filtering	Machine learning Memory bound Fixed-point arithmetic intensive	Dot product (DLP, ILP) Scalar multiplication (DLP, ILP) Vector addition (DLP, ILP) Sigmoid function (ILP)	Dataflow pipelining Datatype customization Communication customization
Optical Flow	Video processing Memory bound Floating-point arithmetic intensive	1D convolution (DLP, ILP) Outer product (DLP, ILP)	Dataflow pipelining Memory customization Communication customization
Binarized Neural Network (BNN) [39]	Machine learning Compute bound Bitwise operation intensive	Binarized 2D convolution (SLP, DLP, ILP) Binarized dot product (SLP, DLP, ILP)	Memory customization Datatype customization Communication customization
Face Detection [25]	Video processing Compute bound Integer arithmetic intensive	Image scaling (DLP, ILP) Cascaded classifiers (DLP, ILP)	Memory customization Datatype customization

Proceed Methodologically

Adjusting C code and pragmas to find the “right” micro-architecture is a major design step...

> 5 Steps to design closure – UG1197 (Chapter 4)

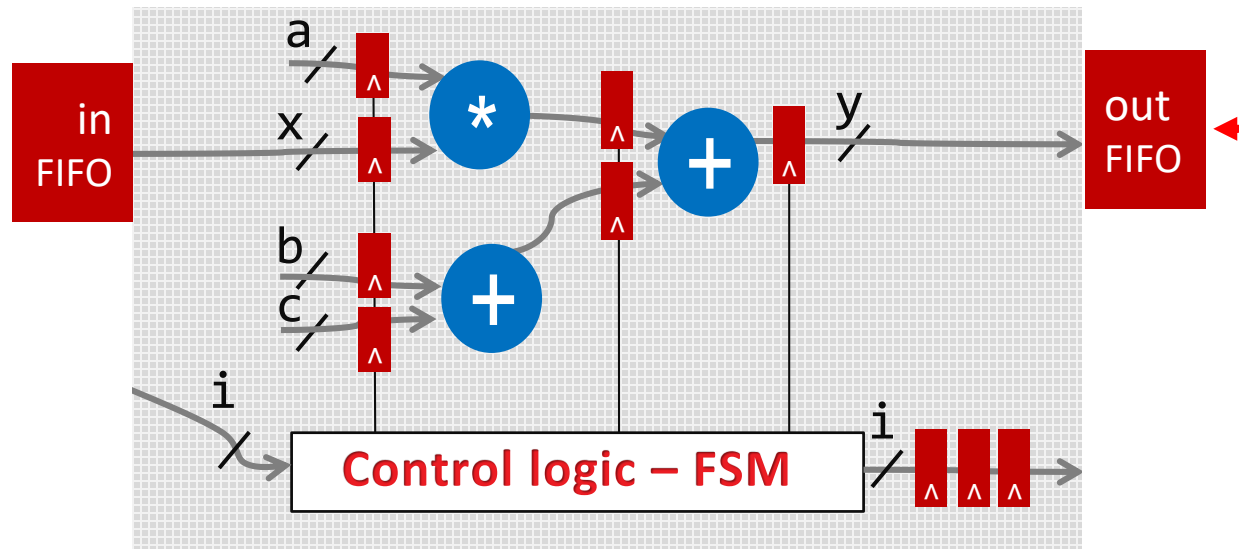
>> The UltraFast High-Level Productivity Design Methodology Guide (Design Hub)



Interface Synthesis

- > Simple code quickly becomes a “real” circuit
 - >> HLS provide block level IO and interface pragma to customize circuit

```
void F (int in[20], int out[20]) {  
    int a,b,c,x,y;  
    for(int i = 0; i < 20; i++) {  
        x = in[i]; y = a*x + b + c; out[i] = y;  
    }  
}
```



The default interface for C arrays (BRAM) can be changed to “FIFO” via a single line pragma (a.k.a directive)...

HLS Adapts Logic to the Design Interface

Apply Instruction Level Parallelism with PIPELINE

> PIPELINE applies to loops or functions

>> Instructs HLS to process variables continuously

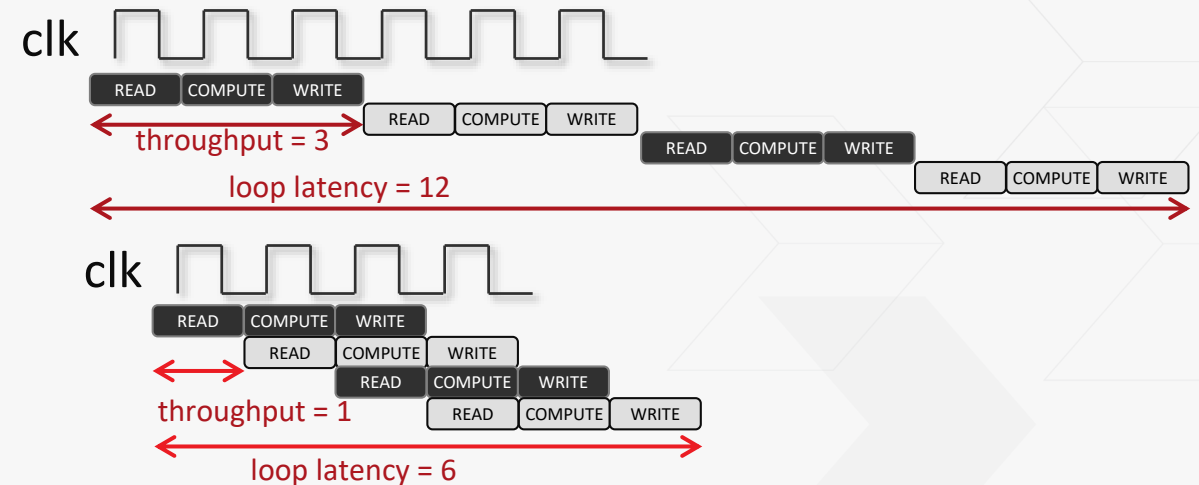
Initiation Interval (II):

Number of clock cycles before the function can accept new inputs

```
void F (...) {  
  ...  
  add: for (i=0;i<=3;i++) {  
    # PRAGMA HLS PIPELINE  
    op_READ;  
    op_COMPUTE;  
    op_WRITE;  
  }  
  ...  
}
```

default

PIPELINE



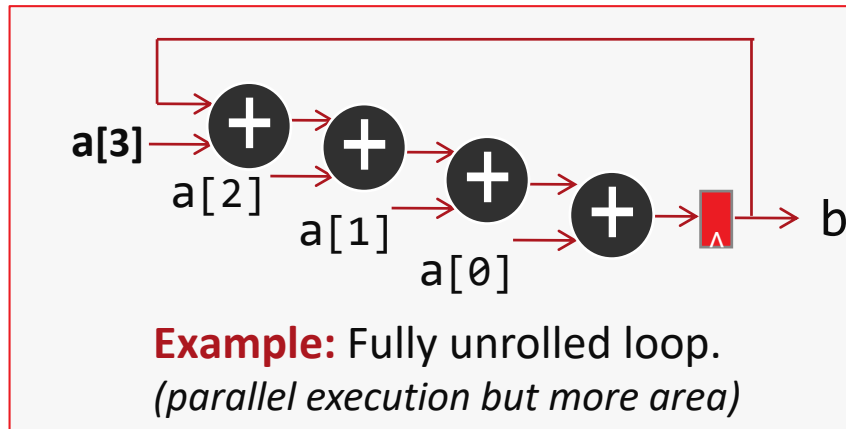
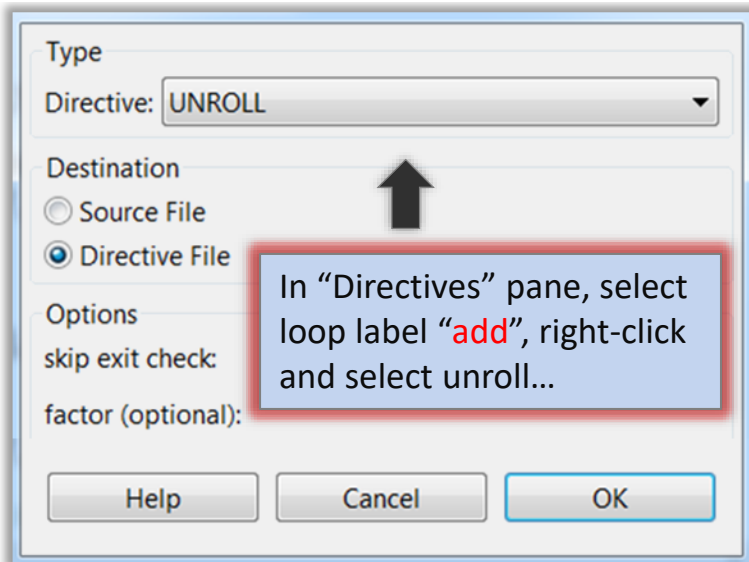
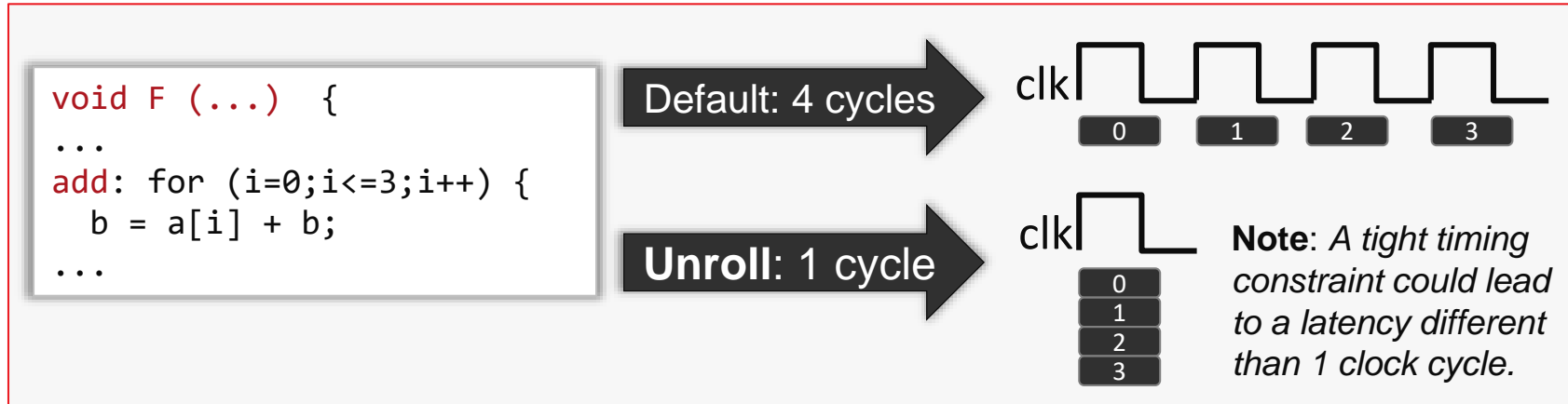
Loop pipelining example

> Allows for loops or functions to process inputs continuously

>> Improves throughput (II gets lower)

Loop Unrolling

> Unroll forces the parallel execution of the instructions in the loop



High performance execution when array elements available in parallel...
Otherwise no benefit from unrolling this loop...

PIPELINE and Automatic Loop Unrolling

Initiation Interval (II):

Number of clock cycles before the function can accept new inputs

> PIPELINE automatically unrolls loops...

```
void fir(data_t x, coef_t c[N], acc_t *y) {  
  
#pragma HLS PIPELINE  
    static data_t shift_x[N];  
    acc_t acc;  
    data_t data;  
  
    acc=0;  
    for (int i=N-1;i>=0;i--) {  
        if (i==0) {  
            shift_x[0] = x;  
            data      = x;  
        } else {  
            shift_x[i] = shift_x[i-1];  
            data      = shift_x[i];  
        }  
        acc+=data*c[i];  
    }  
    *y=acc;  
}
```

QUIZ: Which other pragmas might be useful?

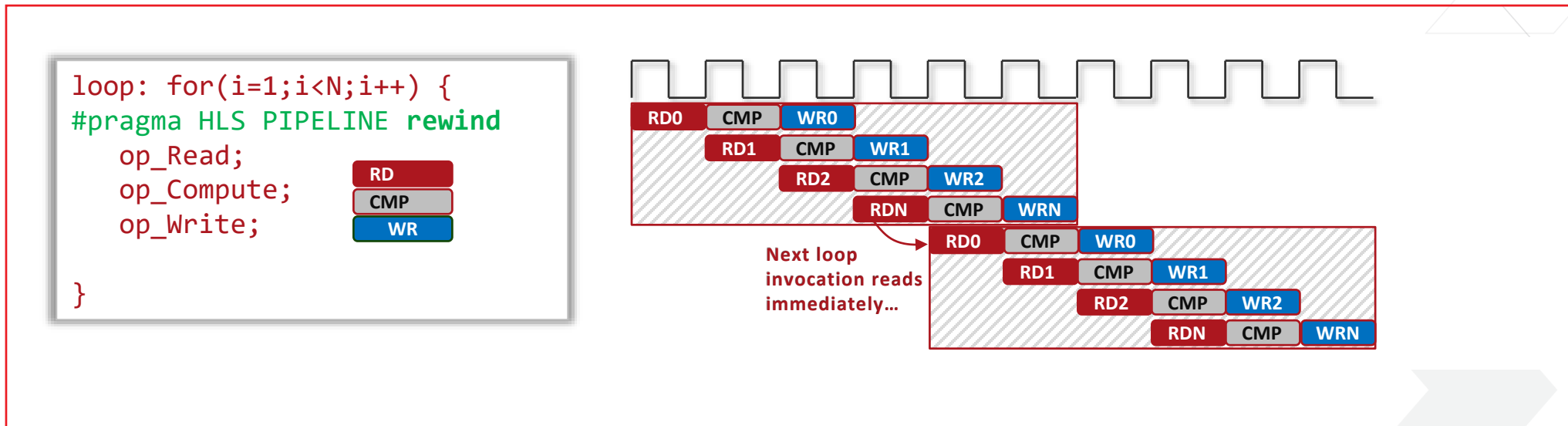
- a) “interface ap_stable” for the coefficients
- b) “array partition” for shift_x
- c) “expression_balance” to control adder tree
- d) *All of the above*

Answer d)

- **ap_stable** helps reduce logic for “c” if the coefficients are expected to be constant
- **Array partitioning** the shifter then ensures all “x” can be accessed in parallel
- **Expression balance** to preserve the inherent multiplier-add cascade chain implied in the C code (longer latency but more efficient once mapped onto DSP blocks)

Removing Inter-Loop Bubbles

- > Rewind for PIPELINE for next loop execution to start as soon as possible
 - >> Removes inter-loop gaps



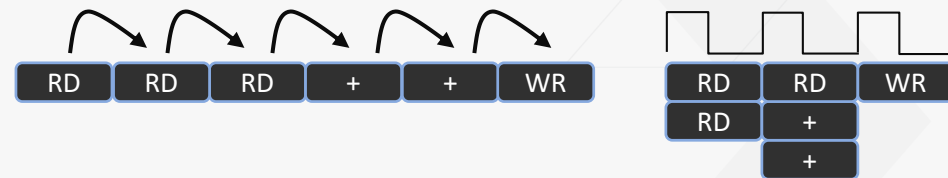
- > See user guide for more information (including the “flush” option)

C Arrays

- > **C Arrays describe memories...**
 - >> Vivado HLS default memory model assumes 2-port BRAMs
- > **Default number of memory ports defined by...**
 - >> How elements of the array are accessed
 - >> The target throughput (a.k.a **initiation interval** also referred to as **II**)

```
void foo (...) {  
    ...  
    SUM_LOOP: for(i=2; i<N; ++i) {  
        sum += mem[i] + mem[i-1] + mem[i-2];  
        ...  
    }  
}
```

See UG902 to get full throughput on this example
• (Chap 3 – Array Accesses and Performance)



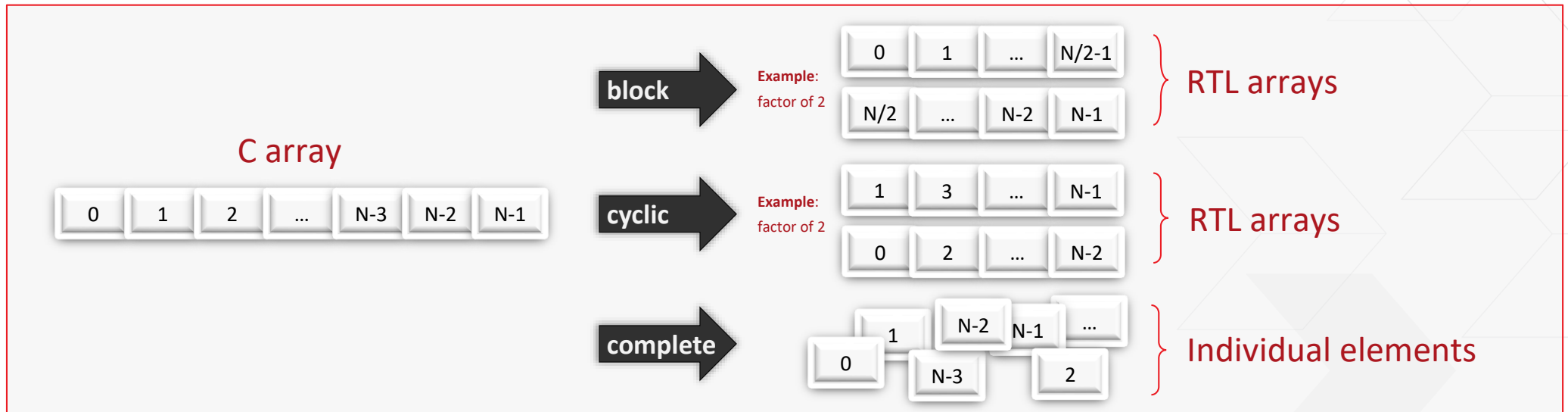
Example: Code implies three reads from a RAM, prevents full throughput

- > **Arrays can be reshaped and/or partitioned to remove bottlenecks**
 - >> Changes to array layout do not require changes to the original code

Partition, Reshape Your C Arrays

> Partitioning splits an array into independent arrays

>> Arrays can be partitioned on any of their dimensions for better throughput



> Reshaping combines array elements into wider containers

>> Different arrays into a single physical memory

>> New RTL memories are automatically generated without changes to C code

Dataflow Pragma – Task Level Parallelism

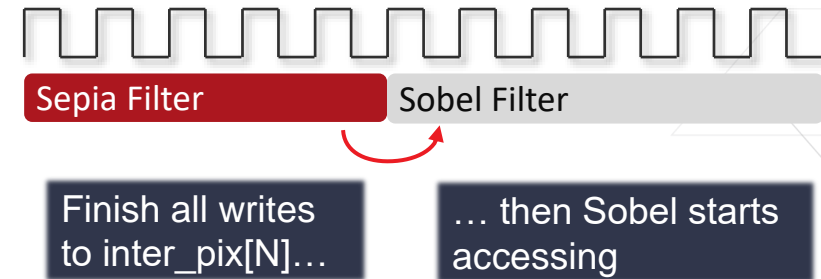
- > By default a C function producing data for another is fully executed first

```
// This memory can be 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);
```

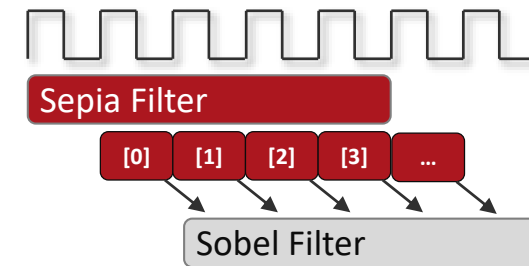
Sepia Filter

Sobel Filter



- > Dataflow allows Sobel to start as soon as data is ready

- >> Functions operate concurrently and continuously
- >> The interval (hence throughput) is improved
- >> Channel buffer has to be filled before consumed for ping-pong



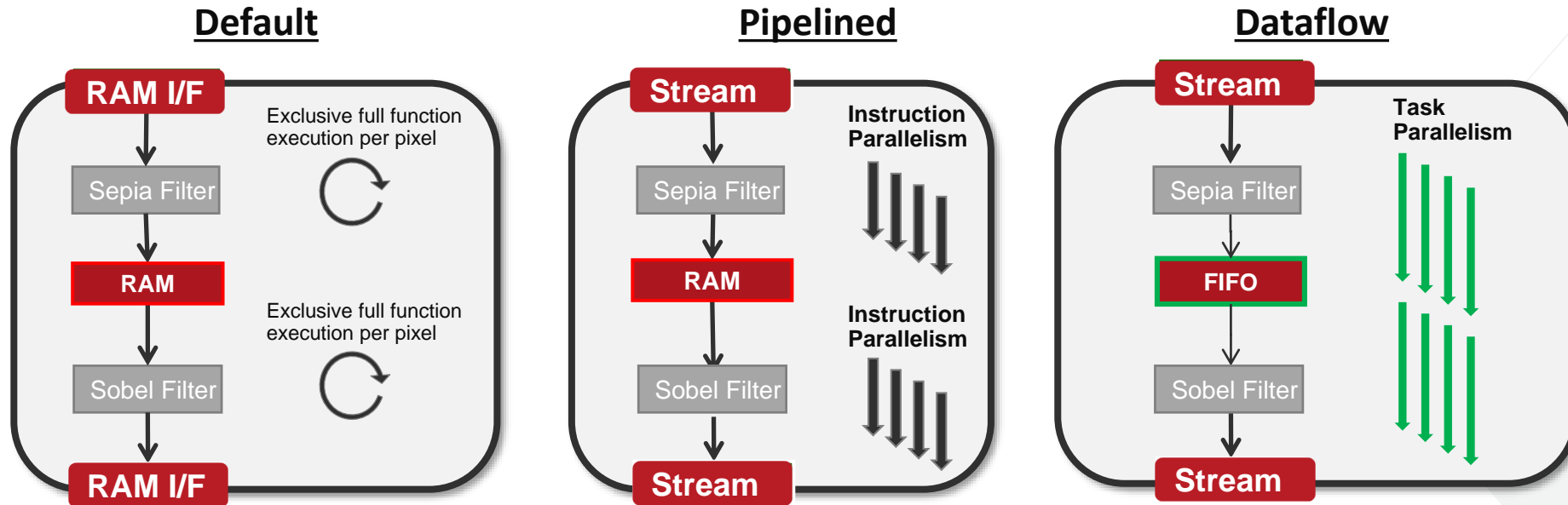
- > Dataflow creates memory channels

- >> Created between loops or functions to store data samples
- >> “Ping-pong” channel holds all the data
- >> “FIFO” for sequential access, no need to store all the data



Video Applications and DATAFLOW

> The FIFO channel with DATAFLOW avoids storing frames between tasks



	<i>Default</i>	<i>Pipelined</i>	<i>Dataflow</i>
BRAM	2792	2790	24
FF	891	1136	883
LUT	2315	2114	1606
Interval (II)	128,744,588	4,150,224	2,076,613

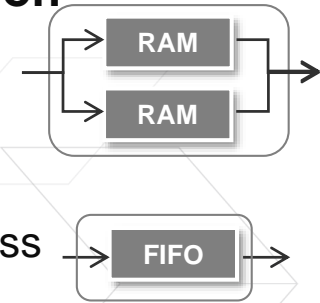
Dataflow Hardware Implementation

- > HLS inserts a “channel” between the functions

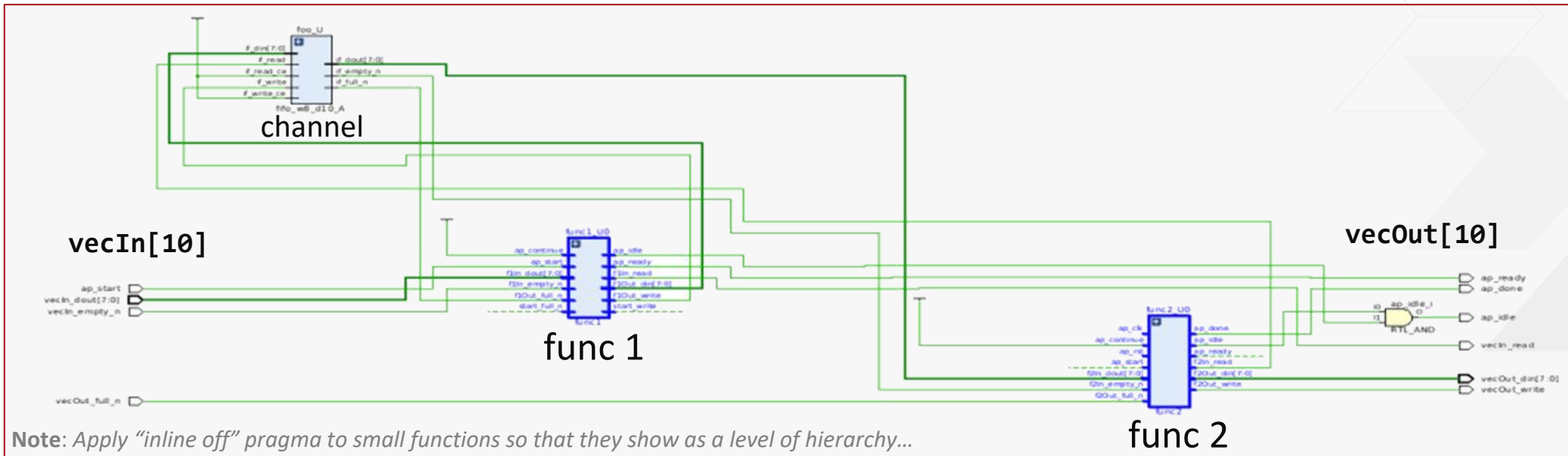


- > Channel implementation

- >> Ping-pong buffer
 - RAM buffers
- >> FIFO
 - Sequential access



- > Vivado implementation (RTL view)



Dataflow Example

- > DATAFLOW allows concurrent execution of two (or more) functions

```
void top(int vecIn[10], int vecOut[10]) {  
#pragma HLS DATAFLOW  
    int tmp[10];  
  
    func1(vecIn,tmp);  
    func2(tmp,vecOut);  
}  
  
void func1(int f1In[10], int f1Out[10]) {  
#pragma HLS INLINE off  
#pragma HLS PIPELINE  
    for(int i=0; i<10; i++) {  
        f1Out[i] = f1In[i] * 10;  
    }  
}  
  
void func2(int f2In[10], int f2Out[10]) {  
#pragma HLS INLINE off  
#pragma HLS PIPELINE  
    for(int i=0; i<10; i++) {  
        f2Out[i] = f2In[i] + 2;  
    }  
}
```

- > Vector I/O are modeled as coming from/to a RAM
- > Code on the left has an II of 5
 - >> i.e. vector size of 10 and 2 elements cycle

Input vector is "BRAM" by default, so only 2 reads in one cycle, hence II is 5

Latency (clock cycles)

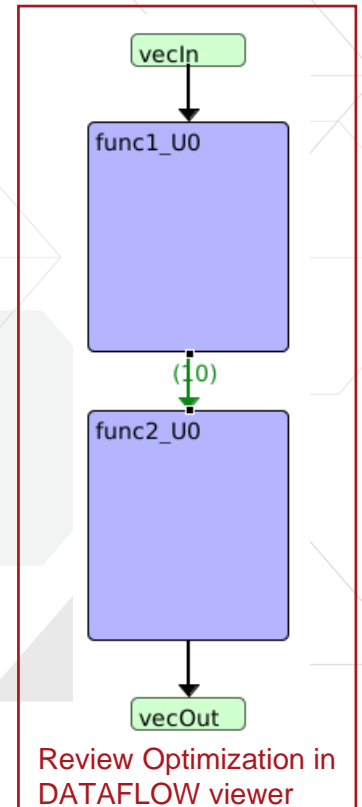
Summary

Latency		Interval		
min	max	min	max	Type
10	10	5	5	dataflow

Detail

Instance

		Latency		Interval		
Instance	Module	min	max	min	max	Type
func1_U0	func1	5	5	5	5	function
func2_U0	func2	4	4	5	5	function



Analyzing Dataflow Results

- > **View simulation waveforms after RTL cosimulation**
 - >> Toolbar button Open Wave Viewer
 - >> Top-level signals in waveform view, pre-grouped into useful bundles

1 Run C/RTL Cosimulation: Vivado Simulator (or Auto)

2 Select Dump Trace "all" or "port"

3 Click OK

4 Click Open Wave Viewer icon

5 Pre-grouped signals:

- Block-level IO
- C inputs
- C outputs

6 Select function:

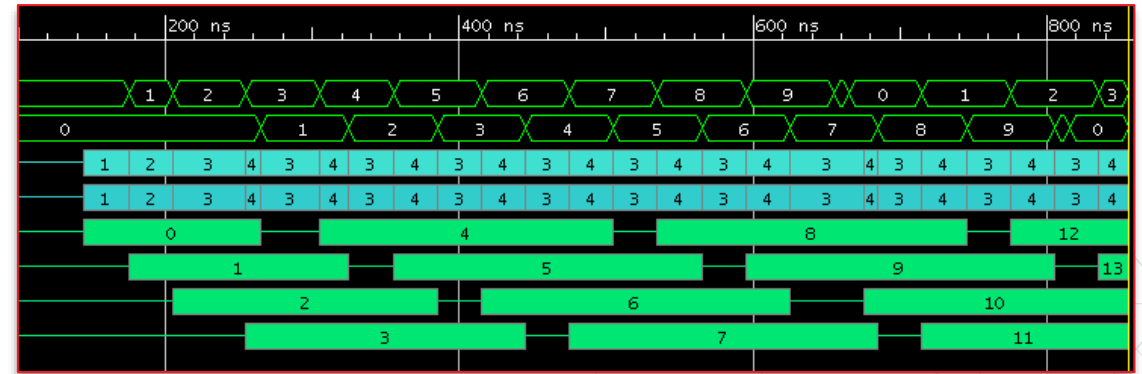
- Add its signals to waveforms
 - ap_done
 - ap_idle
 - ap_ready
 - ap_start

Note: Apply "inline off" pragma to small functions so that they remain a level of hierarchy in HLS...

Analyze Simulation Waveforms

> New Dataflow waveform viewer^(*)

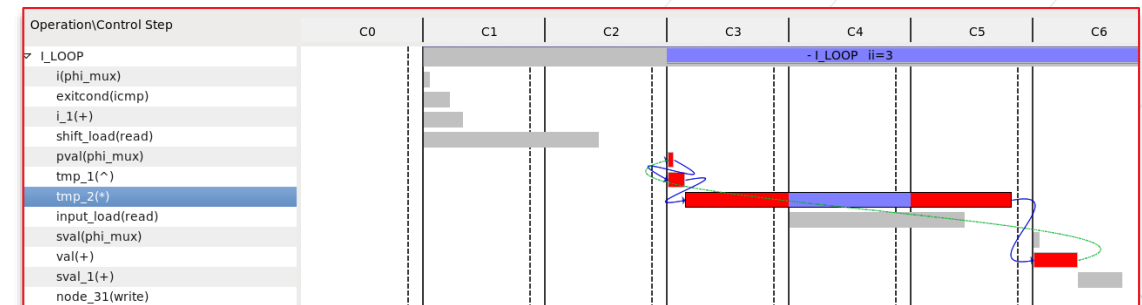
- >> Shows task-level parallelism
- >> Confirm optimizations took place



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

Target Markets for HLS



Aerospace and Defense

- Radar, Sonar
- Signals Intelligence

Communications

- LTE MIMO receiver
- Advanced wireless antenna positioning



Industrial, Scientific, Medical

- Ultrasound systems
- Motor controllers

Audio, Video, Broadcast

- 3D cameras
- Video transport



Automotive

- Infotainment
- Driver assistance

Consumer

- 3D television
- eReaders



Test & Measurement

- Communications instruments
- Semiconductor ATE

Computing & Storage

- High performance computing
- Database acceleration



Vivado HLS Resources

- > Vivado HLS is included in all Vivado HLx Editions (free in WebPACK)
- > Videos on xilinx.com and YouTube
- > DocNav: Tutorials, UG, app notes, videos, etc...
- > Application notes on xilinx.com (also linked from hub)
- > Code examples within the tool itself and on github
- > Instructor led training

QUICK TAKE

 **XILINX**
ALL PROGRAMMABLE.

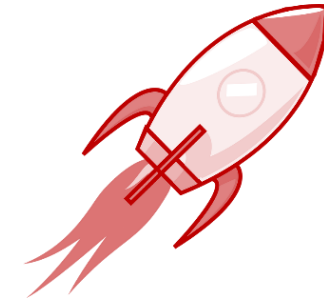
Summary

Performance Boosters for HLS...

- > Compute customization, memory adaptation, datatype optimization

Throughput Optimizations...

- > Apply task and instruction level parallelism



Vivado HLS is not just C synthesis...

- > It's C simulation, automated RTL simulation, interface synthesis, waveform analysis

The logo features a red chevron pointing right, followed by the letters 'XDF' in a bold, white, sans-serif font.

XILINX
DEVELOPER
FORUM

