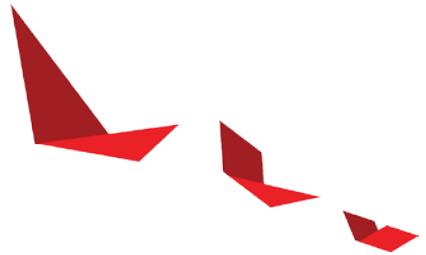
XILINX

Kernel Optimization

Introduction to Vitis

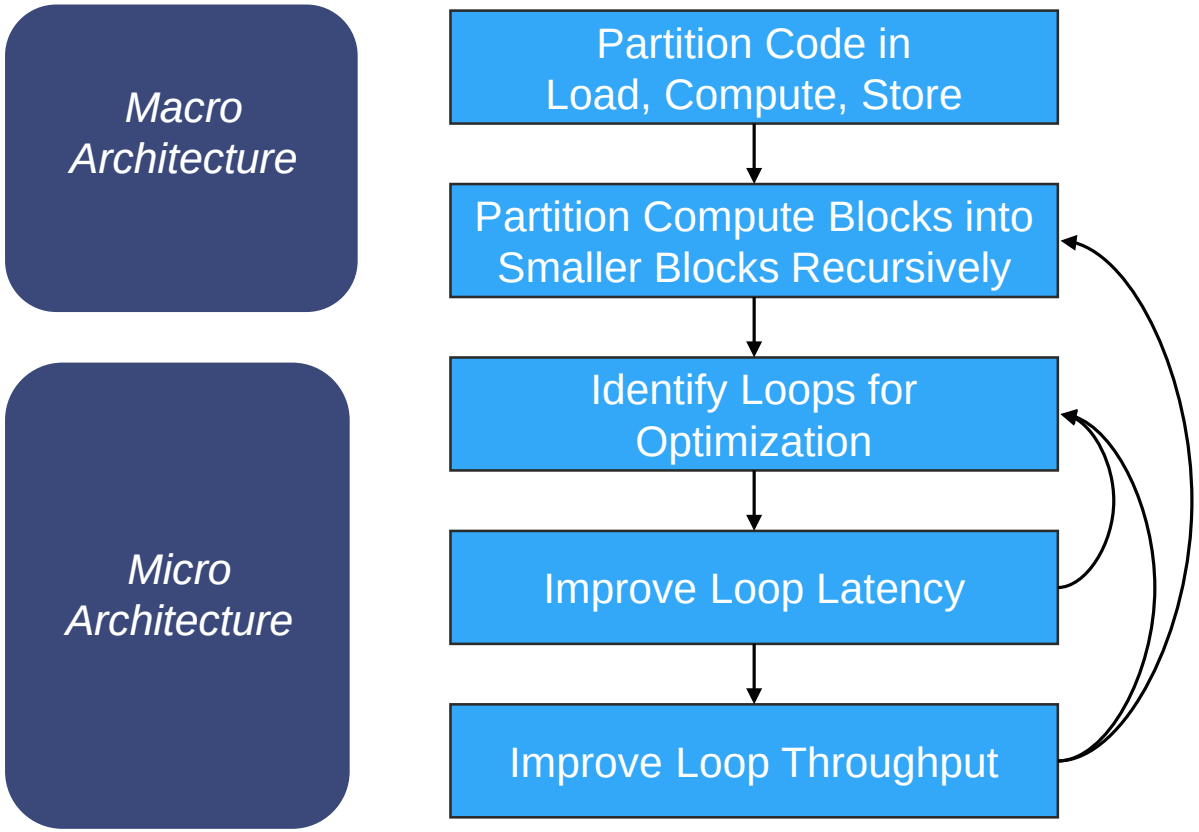
Goal



- Understand C/C++/OpenCL kernel development methodology
- Understand how to optimize kernels
  - Interface Optimizations
  - Unrolling and Pipelining logic
  - Memory optimizations

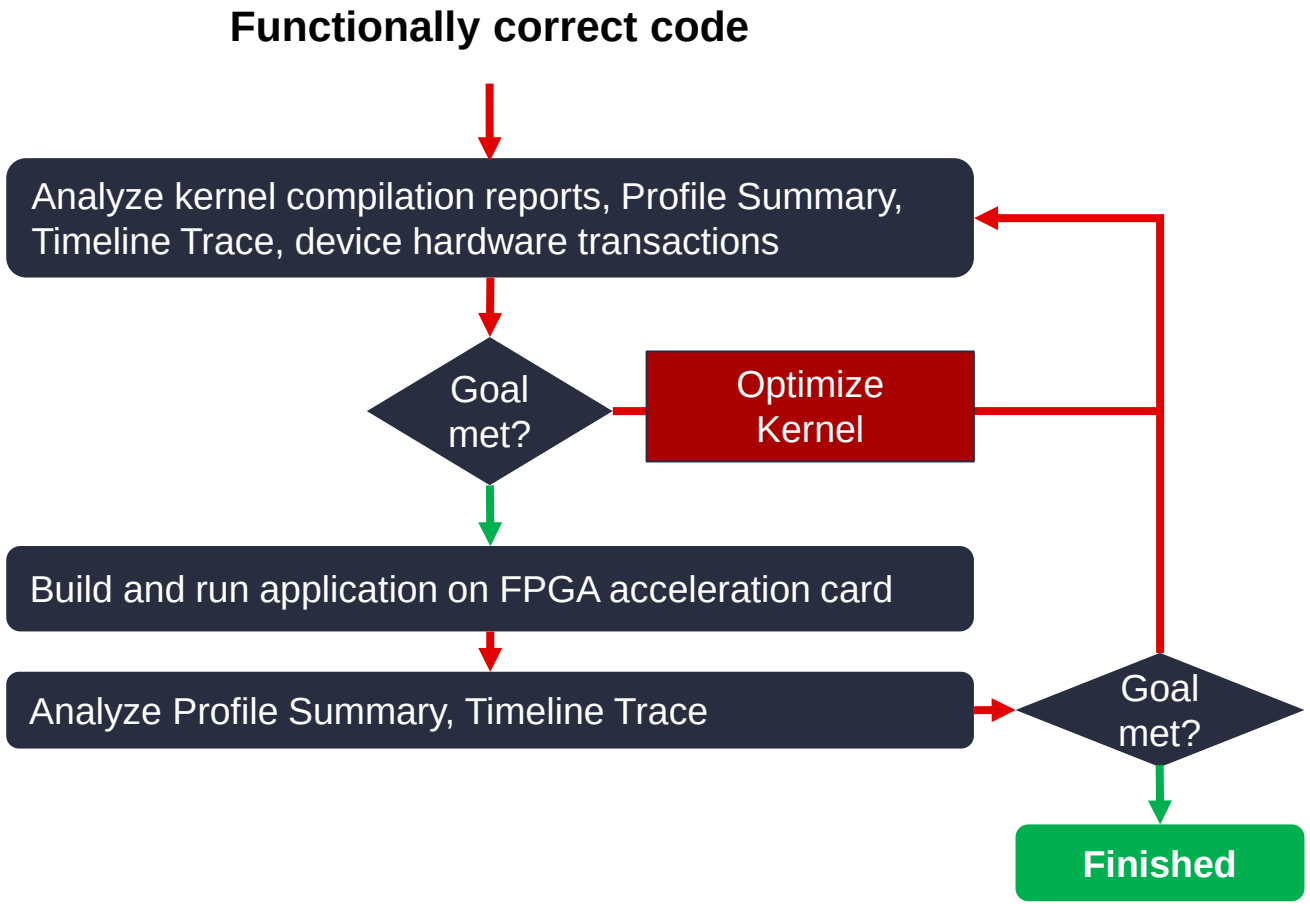
Methodology for Developing C/C++ Kernels

- Key kernel requirements for optimal performance
  - Throughput goal
  - Latency goal
  - Datapath width
  - Number of engines
  - Interface bandwidth



Optimizing Kernel Computation

Hardware Emulation	Run Application and Analyze the Reports
Hardware Run	Run the Application on Hardware and Analyze the Reports



Kernel optimizations

1. Interface Optimization	Using Burst Data Transfers	- Array partitioning technique
	Using Full AXI Data Width	
2. Optimizing Computational Parallelism	Coding Data Parallelism	
	Loop Parallelism – Unrolling Loops; Pipelining Loops	
	Task Parallelism – DATAFLOW	
3. Optimizing Compute Units	Data Width	Macro Operations
	Fixed-Point Arithmetic	Using Optimized Libraries
4. Optimizing Memory Architecture	Array partitioning	

Interface Optimization – Kernel Trace

- Most relevant fields:
  - Burst Length:
    - Describes how any packets are sent
  - Burst Size:
    - Describes the number of bytes

Small burst lengths (as well as burst sizes) are good opportunities to optimize interface performance

Interface Optimization – Using Burst Transfers

- Accessing the global memory bank interface from the kernel has a large latency
- Global memory transfer should be done in bursts
- Pipelining is recommended for burst transfer

```
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
}
```

Burst Transfer

Interface Optimization – Using Full AXI Data Width

Native Data Types

```
void cnn(int *pixel,   // Input pixel
         int *weights, // Input weight matrix
         int *out,     // Output pixel
         ...           // Other input or output ports

#pragma HLS interface m_axi port=pixel offset=slave bundle=gmem
#pragma HLS interface m_axi port=weights offset=slave bundle=gmem
#pragma HLS interface m_axi port=out offset=slave bundle=gmem
```

Arbitrary Data Types

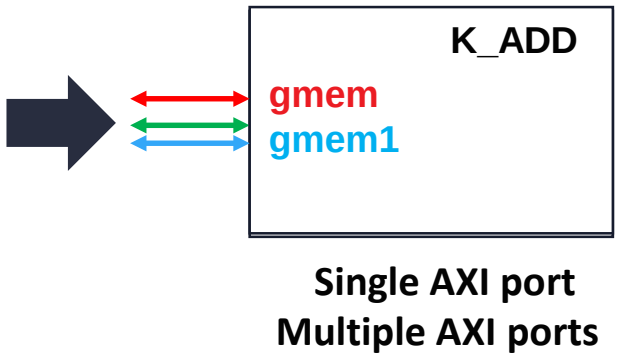
```
void cnn(ap_uint<512> *pixel, // Input pixel
         int *weights, // Input weight matrix
         ap_uint<512> *out,   // Output pixel
         ...                  // Other input or output ports

#pragma HLS interface m_axi port=pixel offset=slave bundle=gmem
#pragma HLS interface m_axi port=weights offset=slave bundle=gmem
#pragma HLS interface m_axi port=out offset=slave bundle=gmem
```

- Datapath width from kernel to memory controller is 512 bits
- For maximum throughput, the full 512 bits should be used
- The kernel code should be modified to take advantage of the full bit width

Interface Bandwidth Optimization – Number of Ports

- Interfaces impact kernel performance
- By default, Vitis tool creates a single **AXI_M** port per kernel
  - All arguments pass through this interface
  - Different I/O processes will have to access the **AXI_M** port sequentially

```
void K_VADD( int *pixel, int *weights, int *out, ...) {
#pragma HLS INTERFACE m_axi port=pixel offset=slave     bundle=gmem
#pragma HLS INTERFACE m_axi port=weights offset=slave   bundle=gmem1
#pragma HLS INTERFACE m_axi port=out offset=slave       bundle=gmem
```

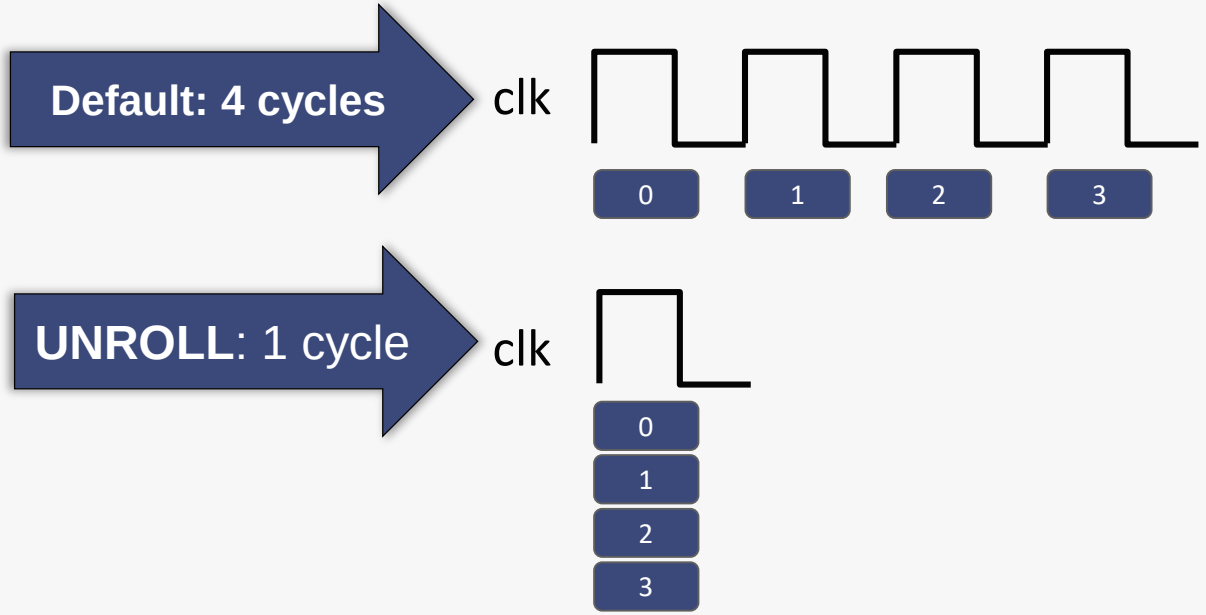


Single AXI port
Multiple AXI ports

Use the “bundle” property on the INTERFACE pragma to create and name AXI_M ports

Optimizing Computational Parallelism – Unroll Loops

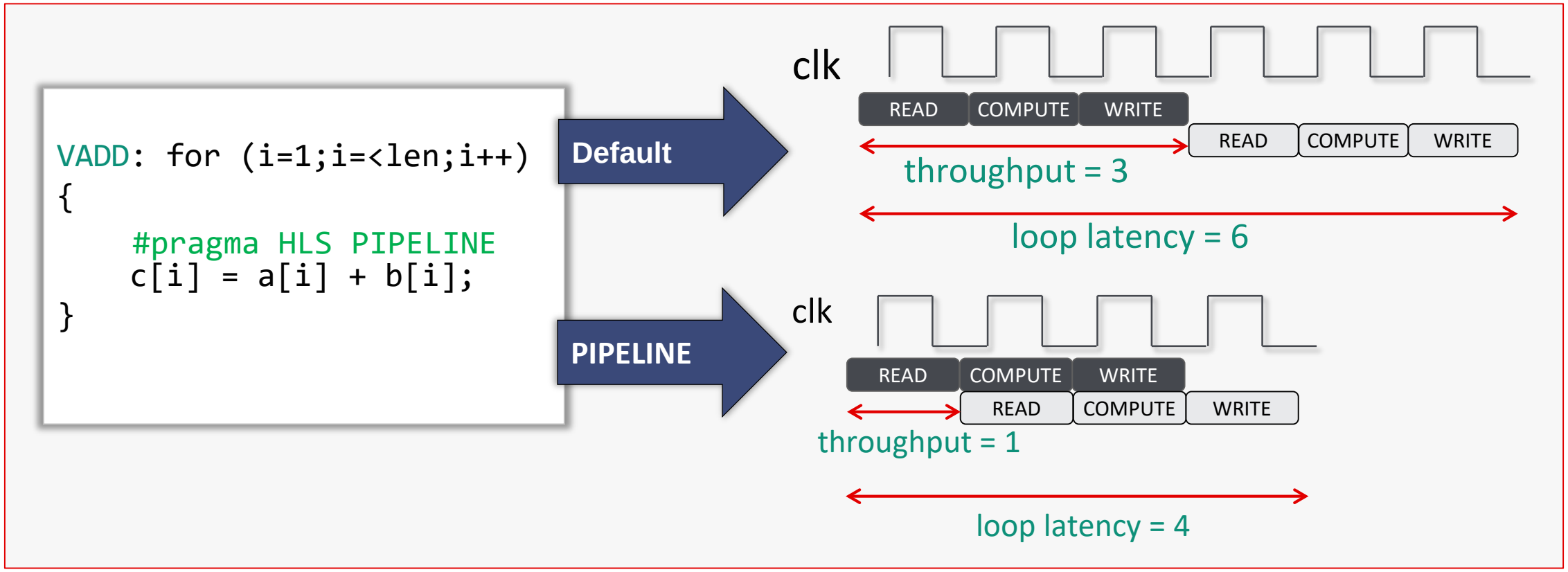
```
void F (...)  {
...
add: for (i=0;i<=3;i++)
{
      #pragma HLS UNROLL
      b = a[i] + c;
...
```



Unroll forces the parallel execution of the instructions in the loop

Optimizing Computational Parallelism – Loop PIPELINE

```
VADD: for (i=1;i=<len;i++)
{
    #pragma HLS PIPELINE
    c[i] = a[i] + b[i];
}
```



The number of cycles it takes to start the next iteration of a loop is called the **initiation interval (II)** of the pipelined loop

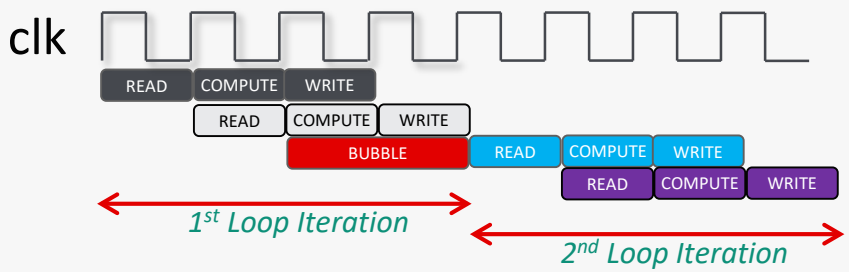
Optimizing Computational Parallelism – Function PIPELINE

```
void F (int A[2],int Z[2]) {

add: for (i=1;i=<2;i++) {
# PRAGMA HLS PIPELINE
      Z[i] = A[i] + 10;
     }
}
```

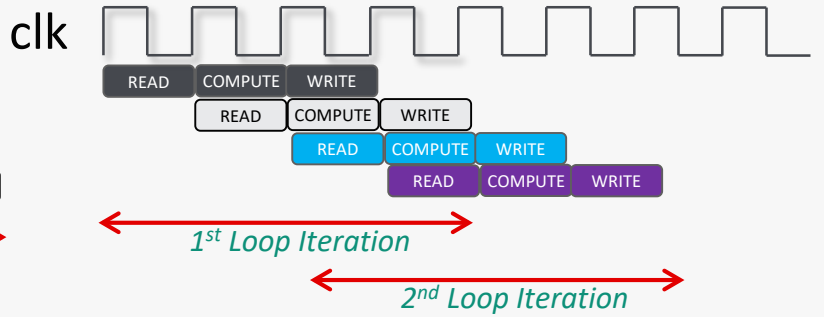
Loop Pipelining

```
void F (int A[2],int Z[2]) {
# PRAGMA HLS PIPELINE

add: for (i=1;i=<2;i++) {
       Z[i] = A[i] + 10;
     }
}
```

Function Pipelining



Loop Pipelining

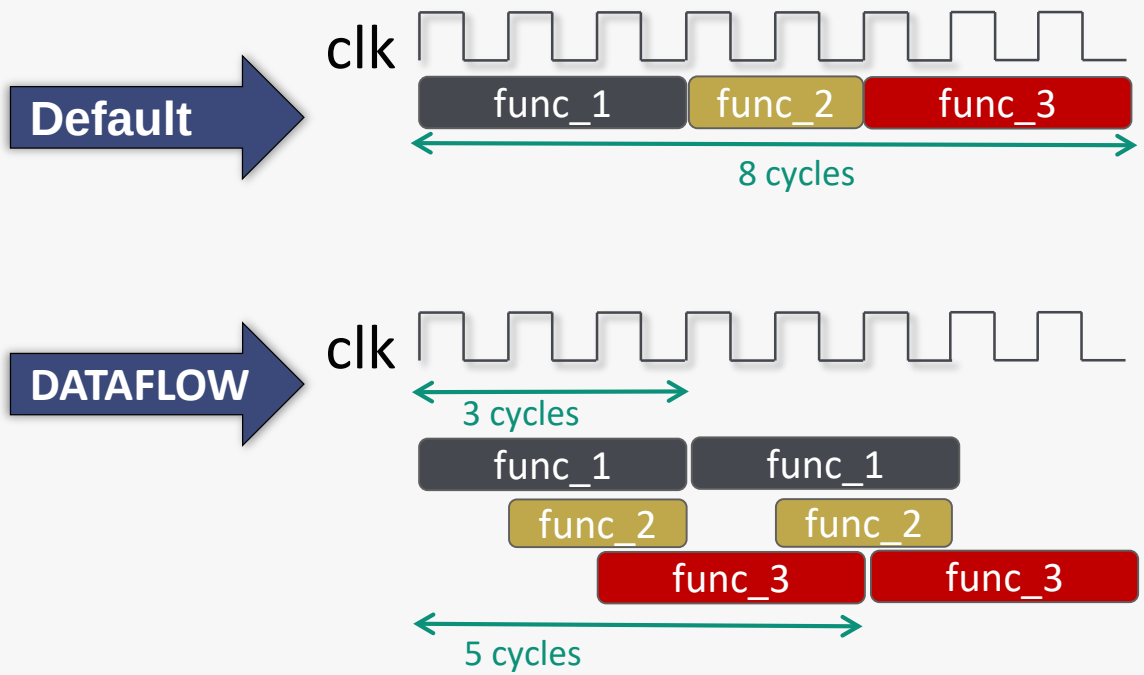


Function Pipelining

XILINX.

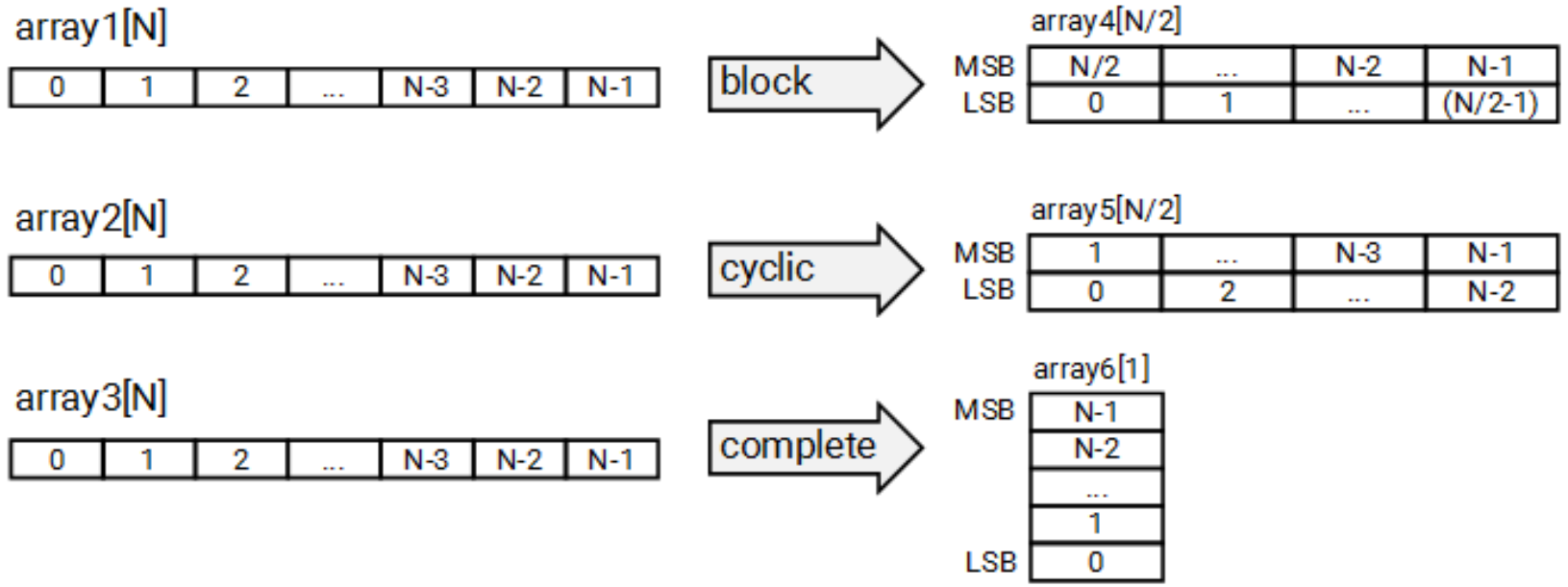
Task Parallelism - DATAFLOW

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

  return d;
}
```



Array Partitioning for Smaller Block RAMs

- Array partitioning can improve performance
- Partition of very big arrays are often a disaster!
  - For multi-dimensional arrays consider applying dimension-based partitioning



Thank You