XDF XILINX DEVELOPER FORUM

Using SDAccel for Host and Accelerator Code Optimizations

Presented By

Peter Frey

October 2, 2018

© Copyright 2018 Xilinx

SDAccel Performance Optimization Agenda

> SDAccel Overview

> Host Code Optimization

> Kernel Code Optimization

> Topological Optimization

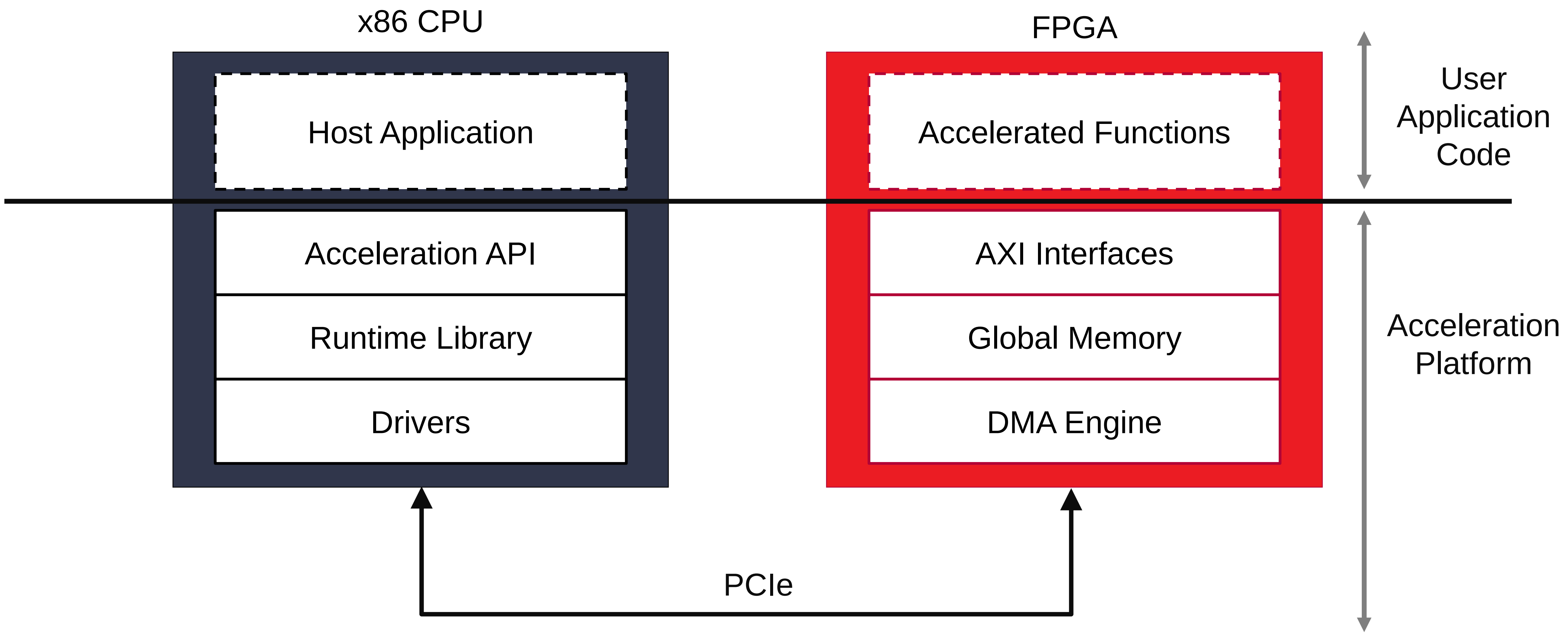
> Implementation Optimization

> Performance Profiling with SDAccel

> Summary

© Copyright 2018 Xilinx

Architecture of an FPGA Accelerated Application



© Copyright 2018 Xilinx

XILINX.

Hardware Acceleration

- **When to USE**
  - Algorithm allows for parallelization
  - Many similar tasks

- **When May Not be beneficial**
  - Small problem size
  - Cost of Host to Device transfers outweighs benefit

- **When NOT beneficial**
  - Little to no parallelism
    - Algorithm is highly sequential over multiple data
    - Tasks are highly dependent

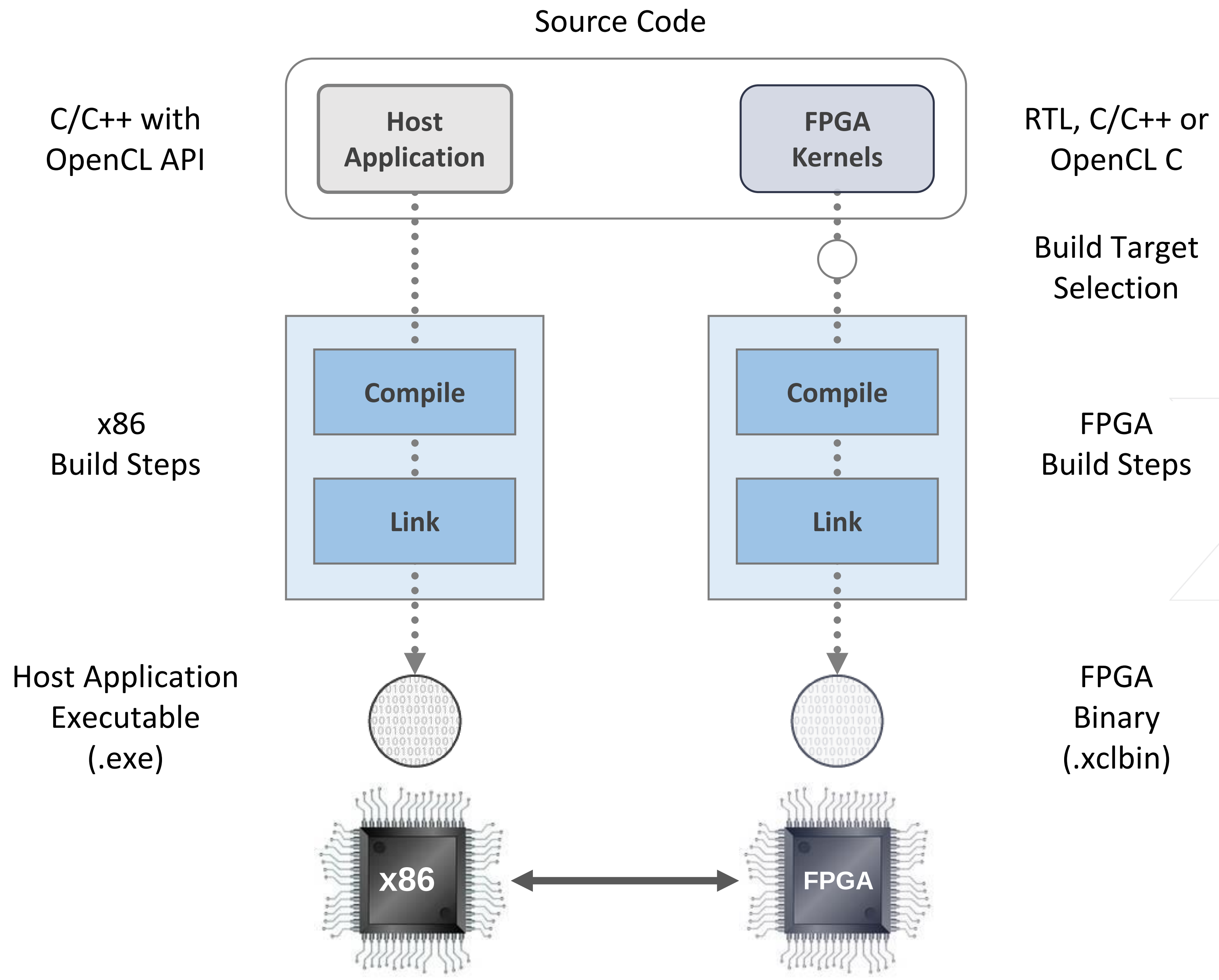
Amdahl’s Law:
If the hardware is 50% of the time, you can accelerate the hardware to zero and you only get 2X

© Copyright 2018 Xilinx

Overview of SDAccel

© Copyright 2018 Xilinx

Flow Overview

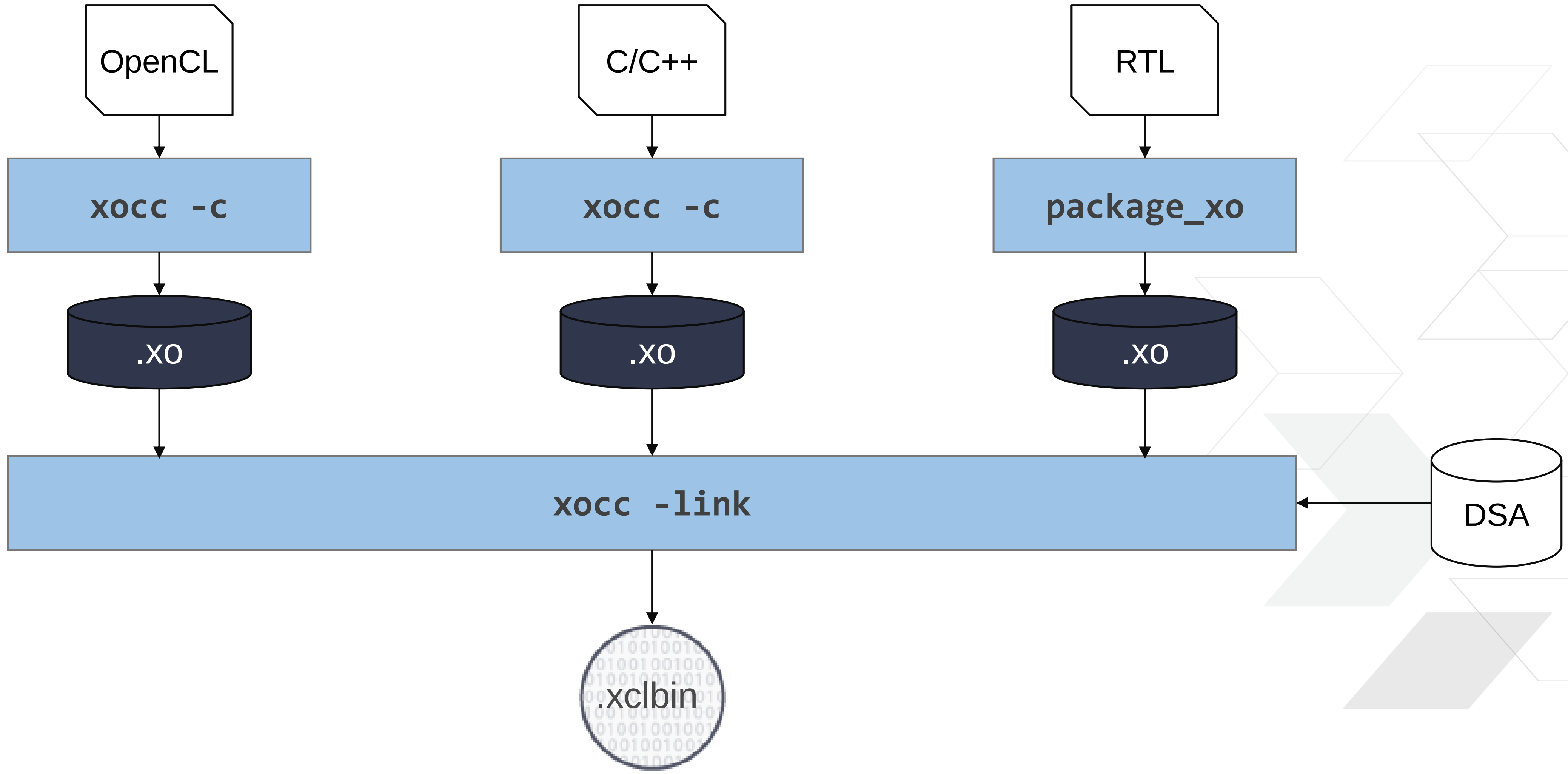


XDF XILINX DEVELOPER FORUM

© Copyright 2018 Xilinx

XILINX

The FPGA Kernel Compilation Flow



© Copyright 2018 Xilinx

SDAccel Execution Modes

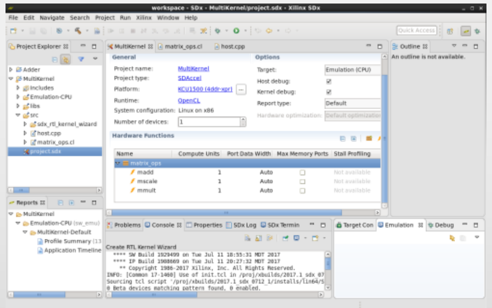
Software Emulation	Hardware Emulation	Hardware Execution
Host application runs with a C/C++ or OpenCL model of the Kernels	Host application runs with a simulated RTL model of the Kernels	Host application runs with actual FPGA implementation of the Kernels
Confirm functional correctness of the system	Test the host / kernel integration, get performance estimates	Confirm system runs correctly and with desired performance
Fastest turnaround time	Best debug capabilities	Accurate performance results

© Copyright 2018 Xilinx

SDAccel Development, Debug & Analysis

- > **Designed to develop and integrate FPGA based acceleration technology into general software solutions**
- > **Fully integrated Eclipse based development environment**
- > **Automatic hardware execution flows support**
- > **Provides software and acceleration debugging capabilities**
- > **Enables detailed system performance analysis**

Develop / Debug


workspace - SDx - MultiKernel/project.sdx - Xilinx SDx
File Edit Navigate Search Project Run Xilinx Window Help
Project Explorer
MultiKernel
matrix_ops.cl
host.cpp
Outline
An outline is not available.
Reports
Problems
Console
Properties
SDx Log
SDx Termin
Target Con
Emulation
Debug

General

Project name: MultiKernel
Project type: SDAccel
Platform: KCU1500 (4ddr-xpr)
Runtime: OpenCL
System configuration: Linux on x86
Number of devices: 1

Options

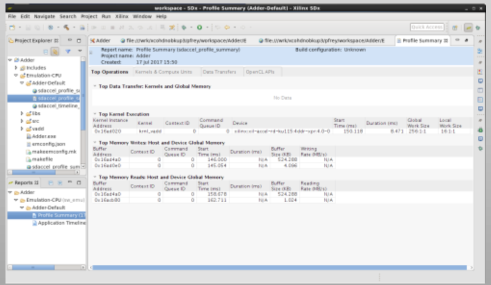
Target: Emulation (CPU)
Host debug: ☑
Kernel debug: ☑
Report type: Default
Hardware optimization: Default optimization

Hardware Functions

Name	Compute Units	Port Data Width	Max Memory Ports	Stall Profiling
matrix_ops				
madd	1	Auto	☐	Not available
mscale	1	Auto	☐	Not available
mmult	1	Auto	☐	Not available

```
Create RTL Kernel Wizard
 **** SW Build 1929499 on Tue Jul 11 18:55:31 MDT 2017
 **** IP Build 1908669 on Tue Jul 11 20:27:32 MDT 2017
   ** Copyright 1986-2017 Xilinx, Inc. All Rights Reserved.
INFO: [Common 17-1460] Use of init.tcl in /proj/xbuilds/2017.1_sdx_07
Sourcing tcl script '/proj/xbuilds/2017.1_sdx_0712_1/installs/lin64/S
0 Beta devices matching pattern found, 0 enabled.
```

Profile Report



Report name: Profile Summary (sdaccel_profile_summary)
Project name: Adder
Created: 17 Jul 2017 15:50
Build configuration: Unknown

Top Data Transfer: Kernels and Global Memory

No Data

Top Kernel Execution

Kernel Instance Address	Kernel	Context ID	Command Queue ID	Device	Start Time (ms)	Duration (ms)	Global Work Size	Local Work Size
0x16ad020	krnl_vadd	0	0	xilinx:xil-accel-rd-ku115:4ddr-xpr:4.0-0	150.118	8.471	256:1:1	16:1:1

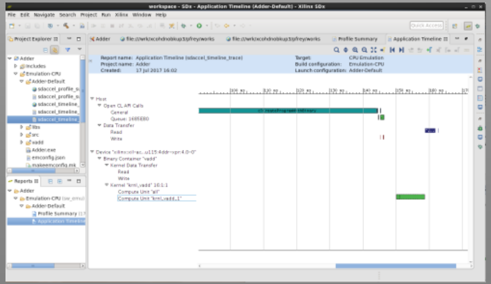
Top Memory Writes: Host and Device Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Writing Rate (MB/s)
0x16ad4a0	0	0	146.000	N/A	524.288	N/A
0x16ad0e0	0	0	145.054	N/A	4.096	N/A

Top Memory Reads: Host and Device Global Memory

Buffer Address	Context ID	Command Queue ID	Start Time (ms)	Duration (ms)	Buffer Size (KB)	Reading Rate (MB/s)
0x16ad4a0	0	0	158.678	N/A	524.288	N/A
0x16adb80	0	0	162.711	N/A	1.024	N/A

Application Timeline


workspace - SDx - Application Timeline (Adder-Default) - Xilinx SDx
File Edit Navigate Search Project Run Xilinx Window Help
Project Explorer
Reports
Profile Summary
Application Timeline
Host
Open CL API Calls
General
Queue: 1685E80
Data Transfer
Read
Write
Binary Container "vadd"
Kernel Data Transfer
Read
Write
Kernel "krnl_vadd" 16:1:1
Compute Unit "all"
Compute Unit "krnl_vadd_1"
clCreateProgramWithBinary

Report name: Application Timeline (sdaccel_timeline_trace)
Project name: Adder
Created: 17 Jul 2017 16:02
Target: CPU Emulation
Build configuration: Emulation-CPU
Launch configuration: Adder-Default

© Copyright 2018 Xilinx

Areas for Performance Optimization

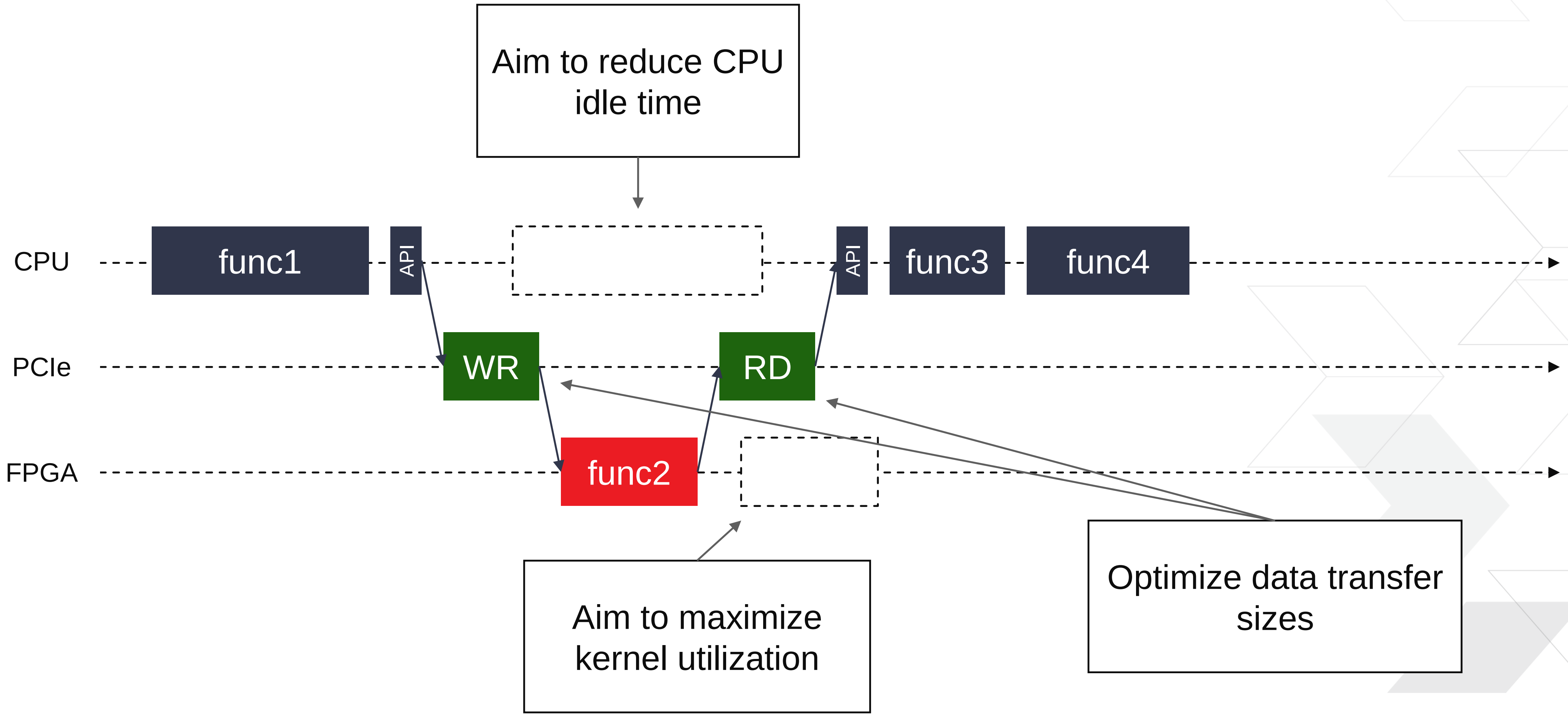
- **Host program optimizations** — host program coding
  - Asynchronous programming, SW pipelining
  - Optimizing transfer sizes
- **Kernel Code optimizations** — kernel program coding
  - Interface Specification (512-bit, bursting interfaces)
  - Dataflow
  - Pipelining
  - Memory Optimization
- **Topological optimizations** — xocc link options
  - Multiple CUs
  - DDR mapping
- **Implementation optimizations** — Vivado options provided to xocc
  - SLR
  - Other Vivado P&R controls

© Copyright 2018 Xilinx

Host Code Optimization

© Copyright 2018 Xilinx

Overview Host Code Optimization

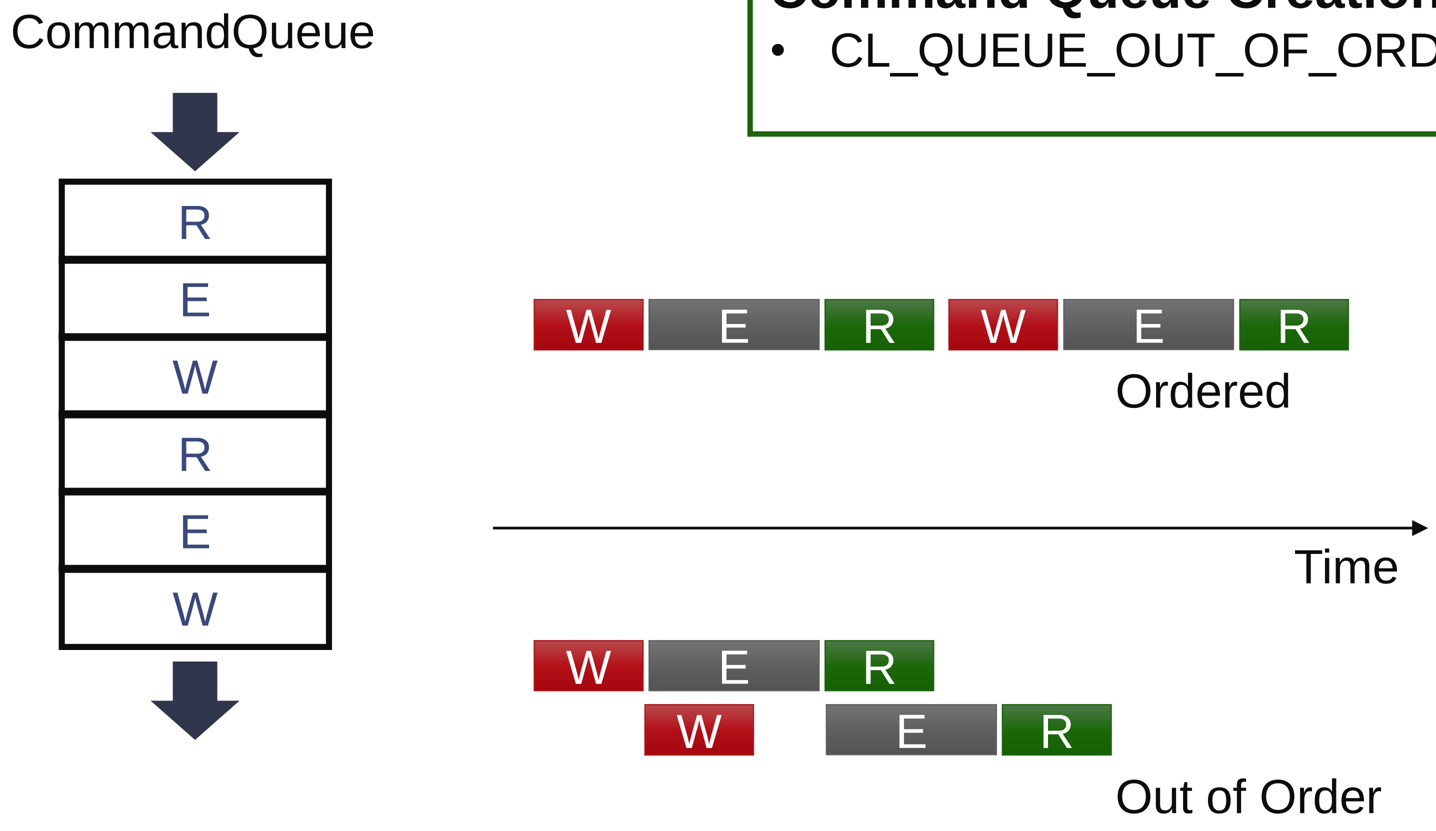

Aim to reduce CPU idle time
CPU
func1
API
API
func3
func4
PCIe
WR
RD
FPGA
func2
Aim to maximize kernel utilization
Optimize data transfer sizes

© Copyright 2018 Xilinx

OpenCL Command Queue Optimization

```
commands = clCreateCommandQueue(context, device_id, CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, &err);
```

Command Queue Creation
- CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE



© Copyright 2018 Xilinx

OpenCL Buffer Allocation and Transfers

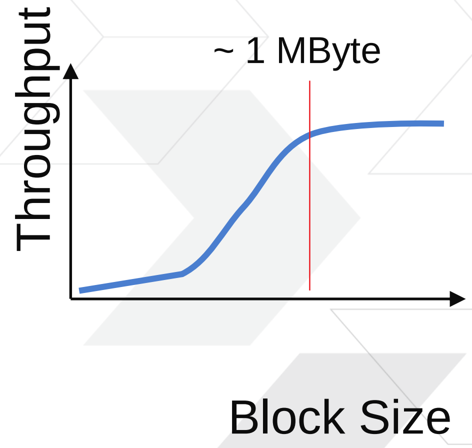
```
cl_mem d_p_A = clCreateBuffer(context,  CL_MEM_READ_WRITE,
                              sizeof(int) * number_of_words, NULL, NULL);
```

> **Buffers are used to exchange data between the host and the device**

> **Aim to reuse available buffers instead of constantly allocating and deallocating new ones**
>> Reduce the overhead of DDR memory management

> **Aim for 1 or 2MBytes transfers**
>> Host ⇔Device effective bandwidth varies with transfer size
>> Allocate optimally sized buffers
>> Group several small buffers in a single transaction

> **READ_WRITE buffer types can create additional dependencies impacting parallel compute unit execution**
>> Only use them when necessary



XDF XILINX DEVELOPER FORUM
© Copyright 2018 Xilinx
XILINX

Task Synchronization

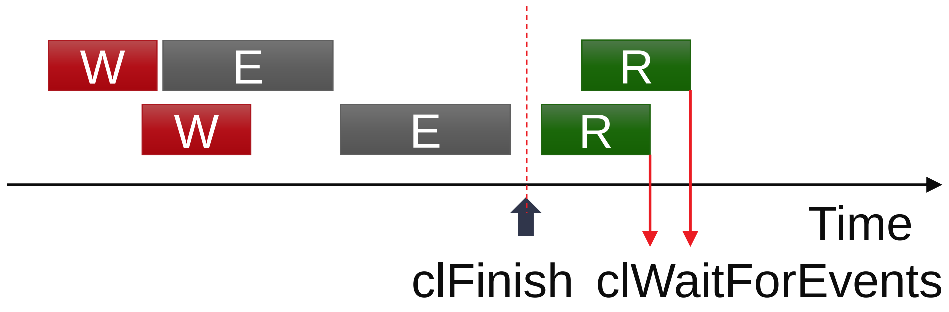
```
for(int i=0; i < 2; i++) {
  d_p_A[i] = clCreateBuffer(…, CL_MEM_READ_ONLY |…,…);
  d_p_B[i] = clCreateBuffer(…, CL_MEM_WRITE_ONLY |…,…);

  clEnqueueMigrateMemObjects(commands, 1, &d_p_A[i], …, 0, NULL, &writeevent[i]);

  clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_p_A[i]);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_p_B[i]);
  clEnqueueTask(commands, kernel, 1, &writeevent[num], 0);
}
clFinish(commands);

clEnqueueMigrateMemObjects(commands, 1, &d_p_B[0], …, 0 , NULL, &readevent[0]);
clEnqueueMigrateMemObjects(commands, 1, &d_p_B[1], …, 0 , NULL, &readevent[1]);

clWaitForEvents(1, &readevent[0]);
clWaitForEvents(1, &readevent[1]);
```

Wait for all events/tasks in Command Queue to finish

Wait for read events to complete

Example:

- One compute unit
- Single Out of Order Command Queue
- Two parallel tasks



XDF XILINX DEVELOPER FORUM

© Copyright 2018 Xilinx

XILINX

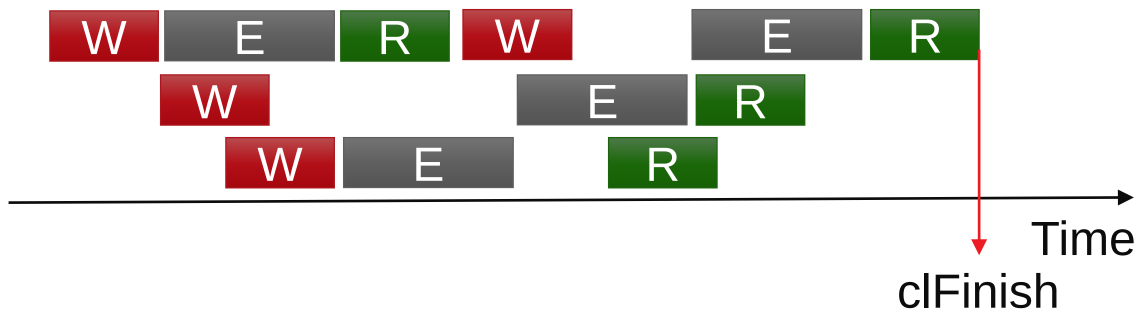
Software Pipelining

```
for(int i=0; i < MAX; i++) {

  d_p_A[i] = clCreateBuffer(…, CL_MEM_READ_ONLY |…,…);
  d_p_B[i] = clCreateBuffer(…, CL_MEM_WRITE_ONLY |…,…);

  clEnqueueMigrateMemObjects(commands, 1, &d_p_A[i], …, 0, NULL, &writeevent[i]);

  clSetKernelArg(kernel, 0, sizeof(cl_mem), &d_p_A[i]);
  clSetKernelArg(kernel, 1, sizeof(cl_mem), &d_p_B[i]);
  clEnqueueTask(commands, kernel, 1, &writeevent[num], &runevent[i]);

  clEnqueueMigrateMemObjects(commands, 1, &d_p_B[i], …, 1, &runevent[i], 0 );
}
clFinish(commands);
```

Schedule all tasks for execute

Use events to synchronize

Wait for all tasks to complete



Example:

- One compute unit
- Single Out of Order Command Queue
- MAX = 4

XDF XILINX DEVELOPER FORUM

© Copyright 2018 Xilinx

XILINX

Kernel Code Optimization

© Copyright 2018 Xilinx

Key Techniques to Develop High Performance C Kernel

1. **Improving Computation efficiency : Parallelize**
   - Customized data type adjusted to requirement
   - Pipeline and Dataflow
   - Unroll (Not always required)
2. **Memory Configuration**
   - Memory customization by array partition
   - Reduce memory access by using local caches, shift registers
3. **Interface and Datatype Optimization**
   - Interface bandwidth consideration
   - Memory Burst Read and Write

© Copyright 2018 Xilinx

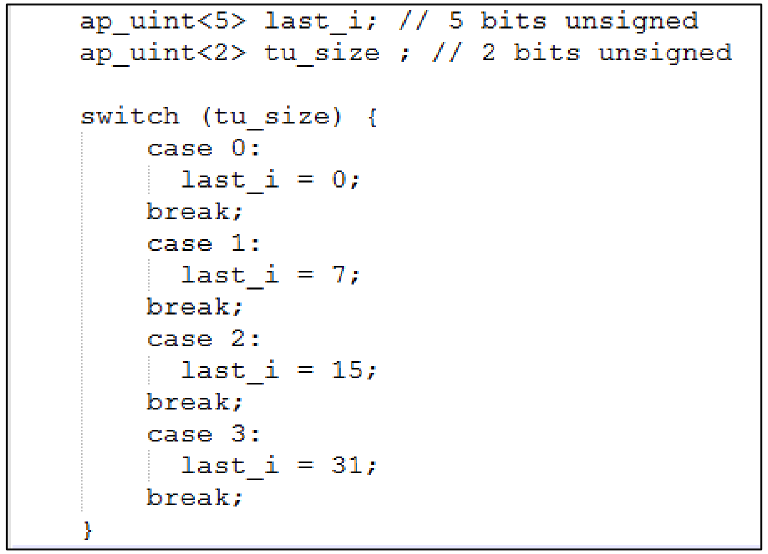
Datawidth Optimization: Bit Accurate Datatypes

> **Leverage Arbitrary precision datatypes from HLS library**

- AP_INT, AP_UINT
- AP_FIXED, AP_UFIXED

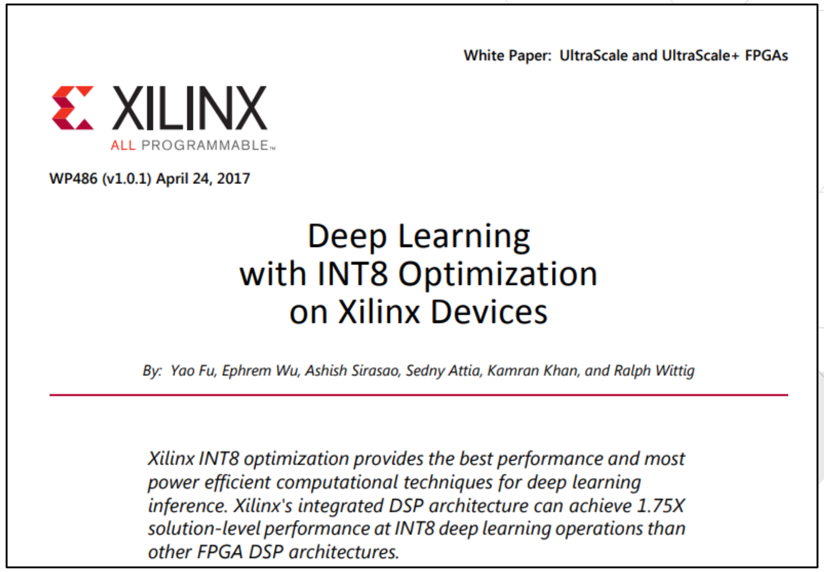
> **Using exact bit width helps to reduce the resource and achieve better performance**

- Practical example: Floating point to Fixed point conversation improve performance
- A whitepaper: Deep Learning with INT8 Optimization



```
ap_uint<5> last_i; // 5 bits unsigned
ap_uint<2> tu_size ; // 2 bits unsigned

switch (tu_size) {
    case 0:
      last_i = 0;
    break;
    case 1:
      last_i = 7;
    break;
    case 2:
      last_i = 15;
    break;
    case 3:
      last_i = 31;
    break;
}
```

Example code shows using bit-accurate integer datatype instead of native short, int etc



White Paper: UltraScale and UltraScale+ FPGAs

XILINX ALL PROGRAMMABLE

WP486 (v1.0.1) April 24, 2017

Deep Learning
with INT8 Optimization
on Xilinx Devices

By: Yao Fu, Ephrem Wu, Ashish Sirasao, Sedny Attia, Kamran Khan, and Ralph Wittig

Xilinx INT8 optimization provides the best performance and most power efficient computational techniques for deep learning inference. Xilinx's integrated DSP architecture can achieve 1.75X solution-level performance at INT8 deep learning operations than other FPGA DSP architectures.

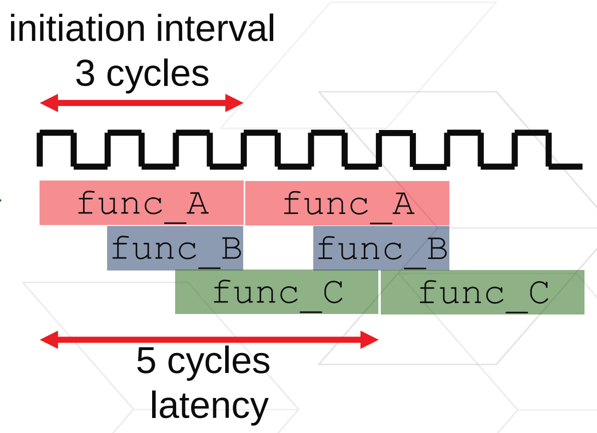
© Copyright 2018 Xilinx

Improve Compute Efficiency: Dataflow

> **Dataflow = Task level parallelism**

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

  return d;
}
```

#pragma HLS dataflow



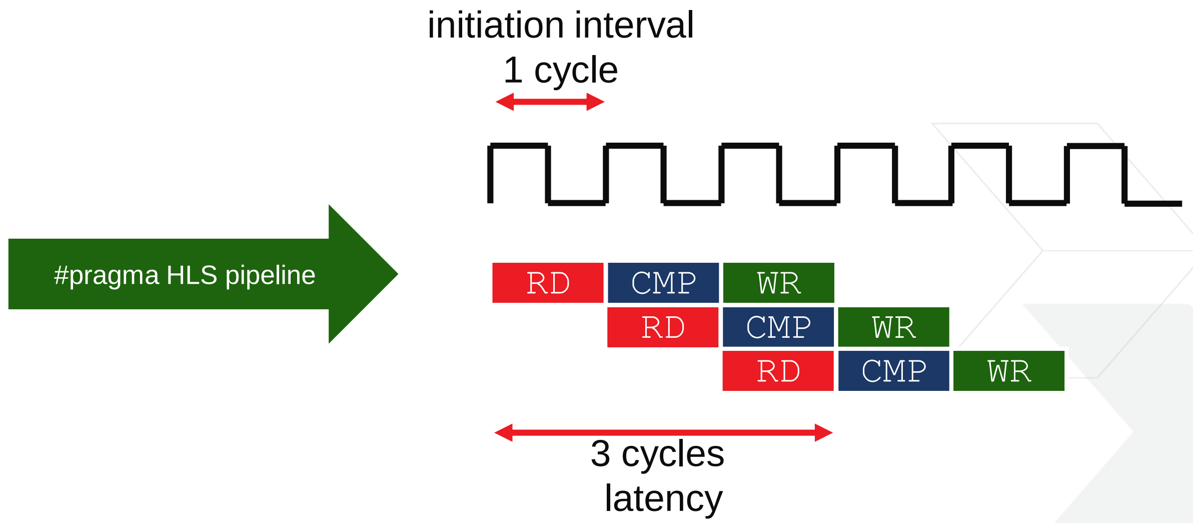
```
top_function(datatype_t * m_in, // Memory d
  datatype_t * m_out, // Memory data Output
  int inp1,      // Other Input
  int inp2) {    // Other Input
#pragma HLS DATAFLOW

hls::stream<datatype_t> in_var1;   // Internal stream to transfer
hls::stream<datatype_t> out_var1;  // data through the dataflow region

read_function(m_in, inp1); // Read function contains pipelined for loop
                           // to infer burst

execute_function(in_var1, out_var1, inp1, inp2); // Core compute function

write_function(out_var1, m_out); // Write function contains pipelined for loop
                                 // to infer burst
}
```

© Copyright 2018 Xilinx

Improve Compute Efficiency: Pipeline

> **Pipeline = Instruction level parallelism**

```
void func(m,n,o) {
  for (int i=2; i>=0; i--) {
    op_read;
    op_compute;
    op_write;
  }
}
```

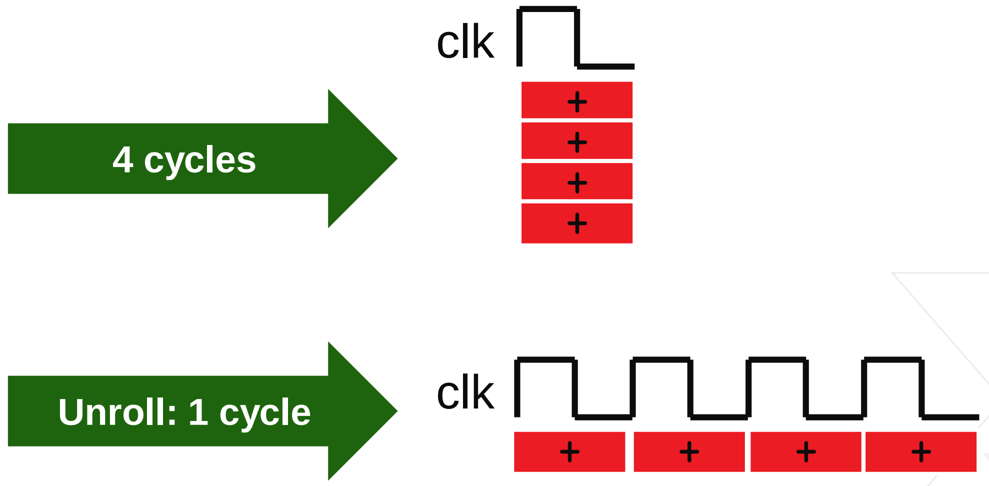


© Copyright 2018 Xilinx

Improve Compute Efficiency: Unrolling Loops

> For smaller body loops with limited number of iterations, unrolling improve performance

```
…
add: for (int i=0; i<=3; i++) {
     b = a[i] + b;
}
…
```



> If complete unrolling is not feasible, exploit partial unrolling

> Unrolling loops with large number of iterations and large body significantly increases resource usage and slows down compilation

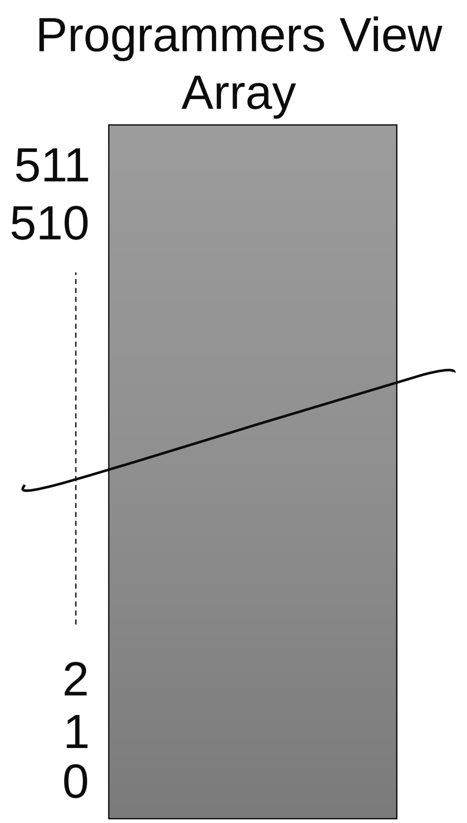
XDF XILINX DEVELOPER FORUM
© Copyright 2018 Xilinx
XILINX

Key Techniques to Develop High Performance C Kernel

1. **Improving Computation efficiency : Parallelize**
   - Customized data type adjusted to requirement
   - Pipeline and Dataflow
   - Unroll (Not always required)
2. **Memory Configuration**
   - Memory customization by array partition
   - Reduce memory access by using local caches, shift registers
3. **Interface and Datatype Optimization**
   - Interface bandwidth consideration
   - Memory Burst Read and Write

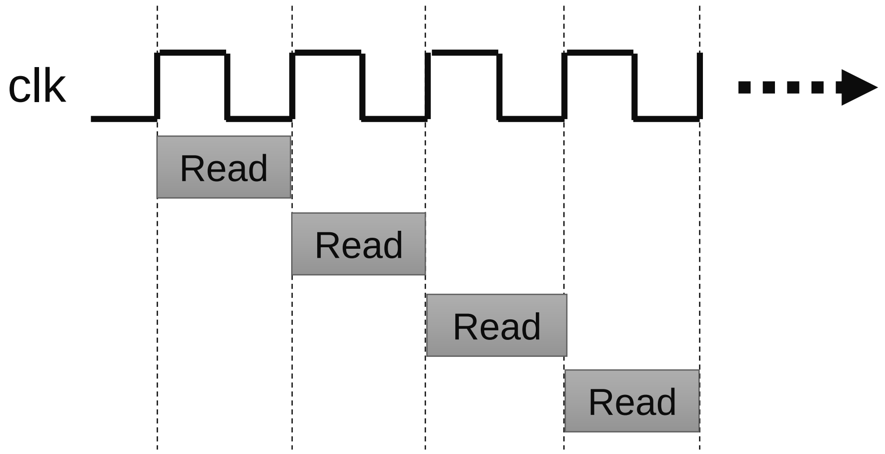
© Copyright 2018 Xilinx

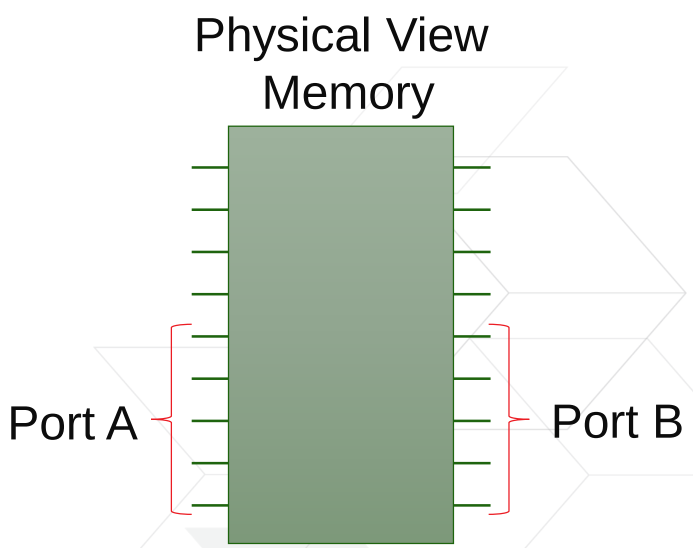
Large Arrays get placed in Memory



```
for ( int i=0; i<512; i++) {
     b = a[i] + b;
}
```

- **Mapping the array ‘a’ to a single memory will force a sequential implementation of the algorithm**





At most 2 Ports each permitting one read per cycle

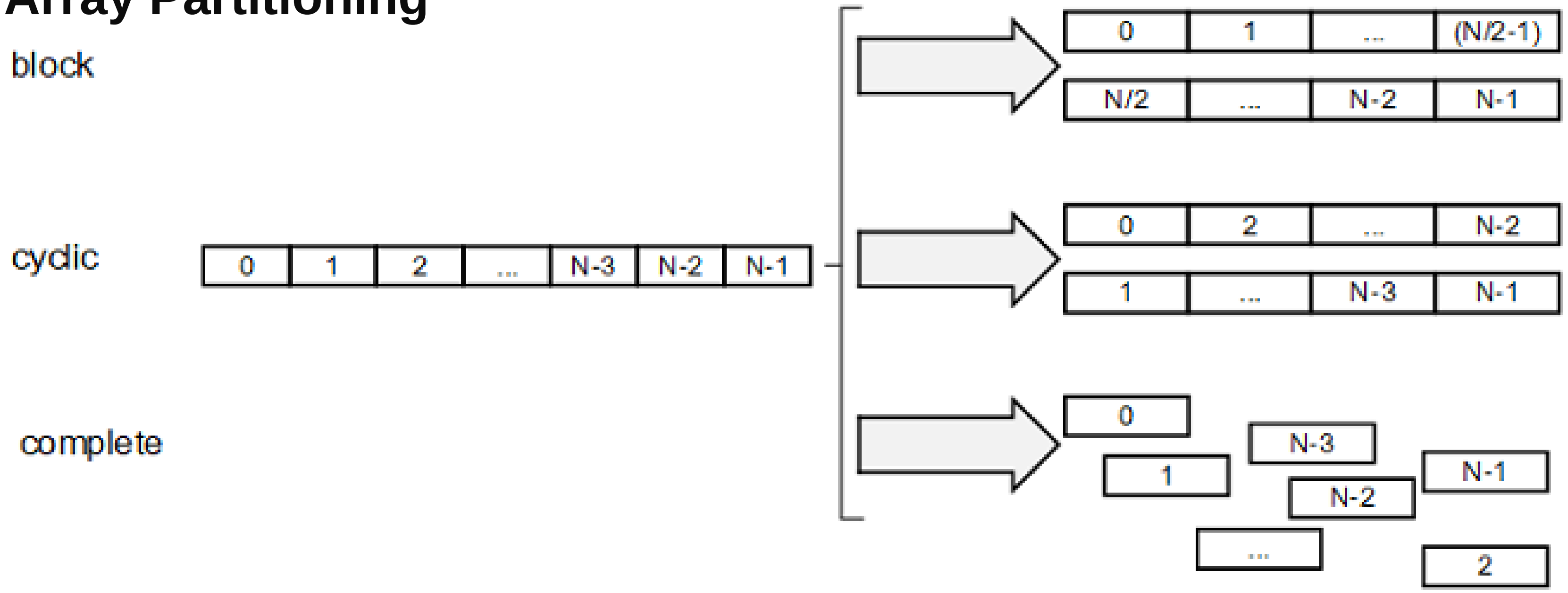
© Copyright 2018 Xilinx

Memory Access crucial for Performance

> Understand the Array access pattern

> Use several BRAMS or Registers to implement arrays (parallel access)

Array Partitioning



© Copyright 2018 Xilinx

Key Techniques to Develop High Performance C Kernel

1. **Improving Computation efficiency : Parallelize**
   - Customized data type adjusted to requirement
   - Pipeline and Dataflow
   - Unroll (Not always required)
2. **Memory Configuration**
   - Memory customization by array partition
   - Reduce memory access by using local caches, shift registers
3. **Interface and Datatype Optimization**
   - Interface bandwidth consideration
   - Memory Burst Read and Write

© Copyright 2018 Xilinx

Interface Bandwidth Optimization – 512 bits

> Kernels use AXI4 MM Master ports to connect with DDR banks over an AXI interconnect

> AXI Interconnect supports up to 512 bit wide transfers

> For maximum throughput, the kernel should use the full 512 bits of the AXI interface

```
void vadd(
        const ap_uint<512> *in1,
        const ap_uint<512> *in2,
        ap_uint<512>       *out,
       int                  size )
{
}
```

Interfaces are 512bit wide

Use ap_uint<512> types to create 512-bit wide AXI_M ports

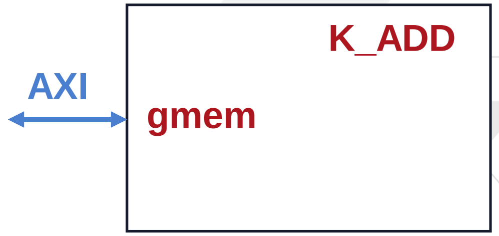
© Copyright 2018 Xilinx

Interface Bandwidth Optimization – Number of Ports

- **Number of AXI_M ports impacts kernel performance**
  - Maximum theoretical bandwidth per AXI_M port is 512bits @ 300MHz (based on platform clock)
- **By default, SDAccel creates a single AXI_M port per kernel**
  - Different I/O processes will have to access the AXI_M port sequentially

```
void K_VADD( dType *A, dType *B, dType*R) {
#pragma HLS INTERFACE m_axi port=A offset=slave
#pragma HLS INTERFACE m_axi port=B offset=slave
#pragma HLS INTERFACE m_axi port=R offset=slave
```

By default, SDAccel maps all pointer arguments to the same AXI_M interface



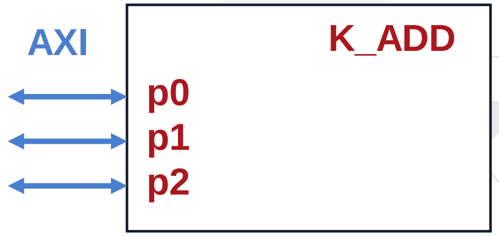
Single AXI_M port

Interface Bandwidth Optimization – Number of Ports

- **Number of AXI_M ports impacts kernel performance**
  - Maximum theoretical bandwidth per AXI_M port is 512bits @ 300MHz (based on platform clock)
- **By default, SDAccel creates a single AXI_M port per kernel**
  - Different I/O processes will have to access the AXI_M port sequentially
- **Adding extra AXI_M ports increases kernel bandwidth**
  - With at least two ports, a kernel can read inputs and write outputs simultaneously

```
void K_VADD( dType *A, dType *B, dType*R) {
#pragma HLS INTERFACE m_axi port=A offset=slave bundle=p0
#pragma HLS INTERFACE m_axi port=B offset=slave bundle=p1
#pragma HLS INTERFACE m_axi port=R offset=slave bundle=p2
```

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



Multiple AXI_Master ports

XDF XILINX DEVELOPER FORUM
© Copyright 2018 Xilinx
XILINX

Interface Bandwidth Optimization – Bursting

- Read/Write accesses to DDR cause have a long latency overhead
- Random sequences of individual accesses are bad for performance
- Bursting is the most efficient way to access DDR as it hides latency
- To ensure bursting behavior, create a dedicated dataflow function in which a pipelined loop reads or writes from an AXI_M port

```
template<typename out_t>
void read_blocks(const out_t *in, hls::stream<out_t> &out, unsigned int blocks) {
  for(unsigned int i = 0; i < blocks*2; i++) {
    #pragma HLS loop_tripcount min=2048 max=2048
    #pragma HLS PIPELINE
    out.write(in[i]);
  }
}
```

Enable burst transfers from global memory

Sequential data access enables streaming data between blocks

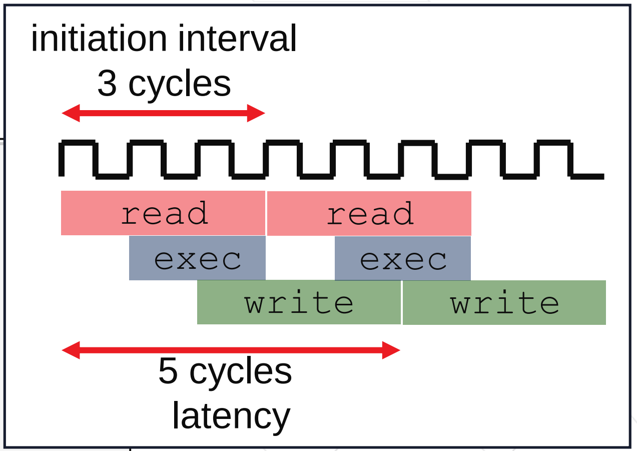
© Copyright 2018 Xilinx

Sustaining Interface Throughput in the Kernel

> **Top function with Read, Compute, Write Dataflow blocks**

```
top_function(datatype_t * m_in, // Memory data Input
  datatype_t * m_out, // Memory data Output
  int inp1,     // Other Input
  int inp2) {   // Other Input
#pragma HLS DATAFLOW

hls::stream<datatype_t> in_var1;   // Internal stream to transfer
hls::stream<datatype_t> out_var1;  // data through the dataflow region

read_function(m_in, inp1); // Read function contains pipelined for loop
                           // to infer burst

execute_function(in_var1, out_var1, inp1, inp2); // Core compute function

write_function(out_var1, m_out); // Write function contains pipelined for loop
                                 // to infer burst
}
```

#pragma HLS dataflow



Concurrent execution of read, execute, write

© Copyright 2018 Xilinx

Topological Optimizations

© Copyright 2018 Xilinx

Multiple Compute Units (Kernel Instances)

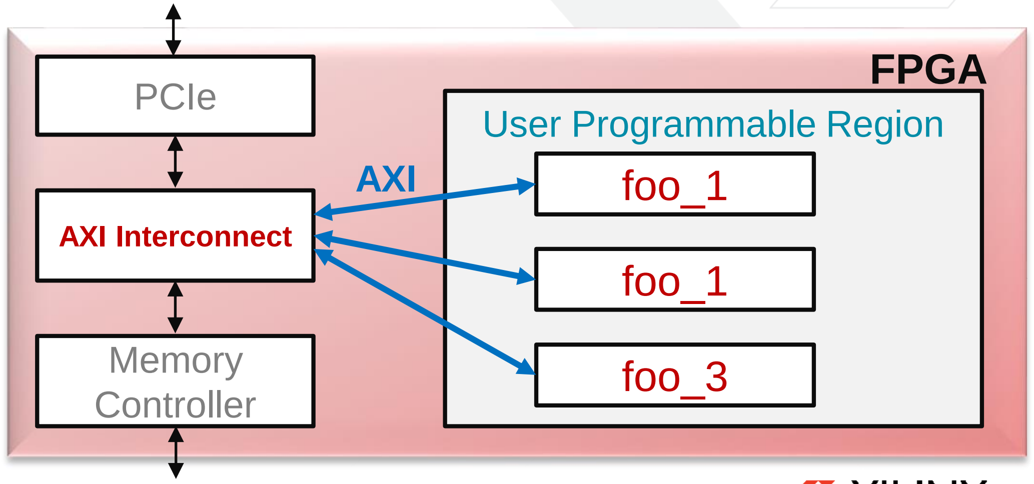
> By default, SDAccel generates 1 instance of each kernel

> Use multiple instances when the same function is performed on independent blocks of data (data-level parallelism)

> Example: 2D Image Filter
  - Use 1 CU to process Y, U and V color planes sequentially
  - Use 3 CUs to process Y, U and V color planes parallel
  - Use 6 CUs to process two images in parallel

```
# Create 3 CUs for kernel “foo”
xocc –l --nk foo:3 <other options>
```

Use the xocc --nk option during the link phase to specify number of CUs for each kernel



© Copyright 2018 Xilinx

Kernel Port Connections to DDR Banks

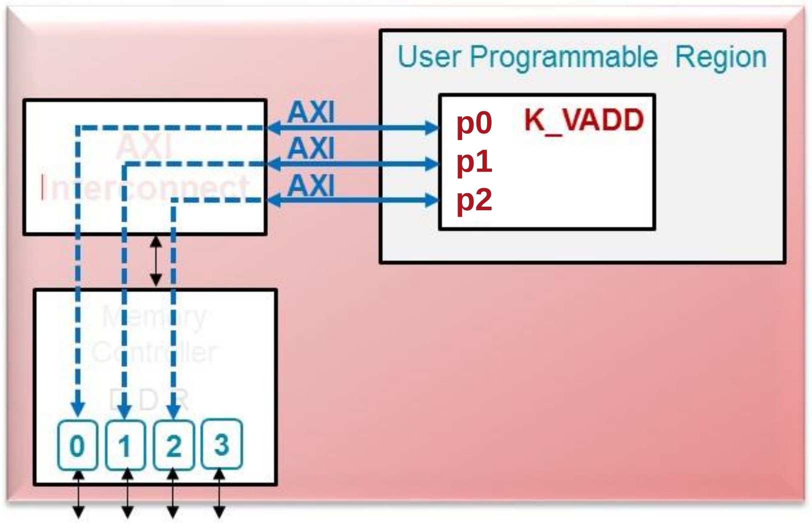
> **SDAccel platforms typically contain 4 DDR banks**

> **By default all kernel AXI_M ports are mapped to the same DDR bank**
>> DDR bandwidth is shared, multiple AXI requests are arbitrated

> **Careful mapping of kernel ports to specific DDR banks improves performance**
>> Take advantage of full DDR bandwidth
>> Simultaneous transfers to each DDR
>> Physical proximity of kernel and DDR improves Fmax

```
xocc -l --sp kernel_top_1.m_axi_p0.bank0 \
        --sp kernel_top_1.m_axi_p1.bank1 \
        --sp kernel_top_1.m_axi_p2.bank2 \
        <other options>
```

Use the xocc --sp option during the link phase to specify desired mapping
Update OpenCL Buffers properties in the host program



Implementation Optimization

© Copyright 2018 Xilinx

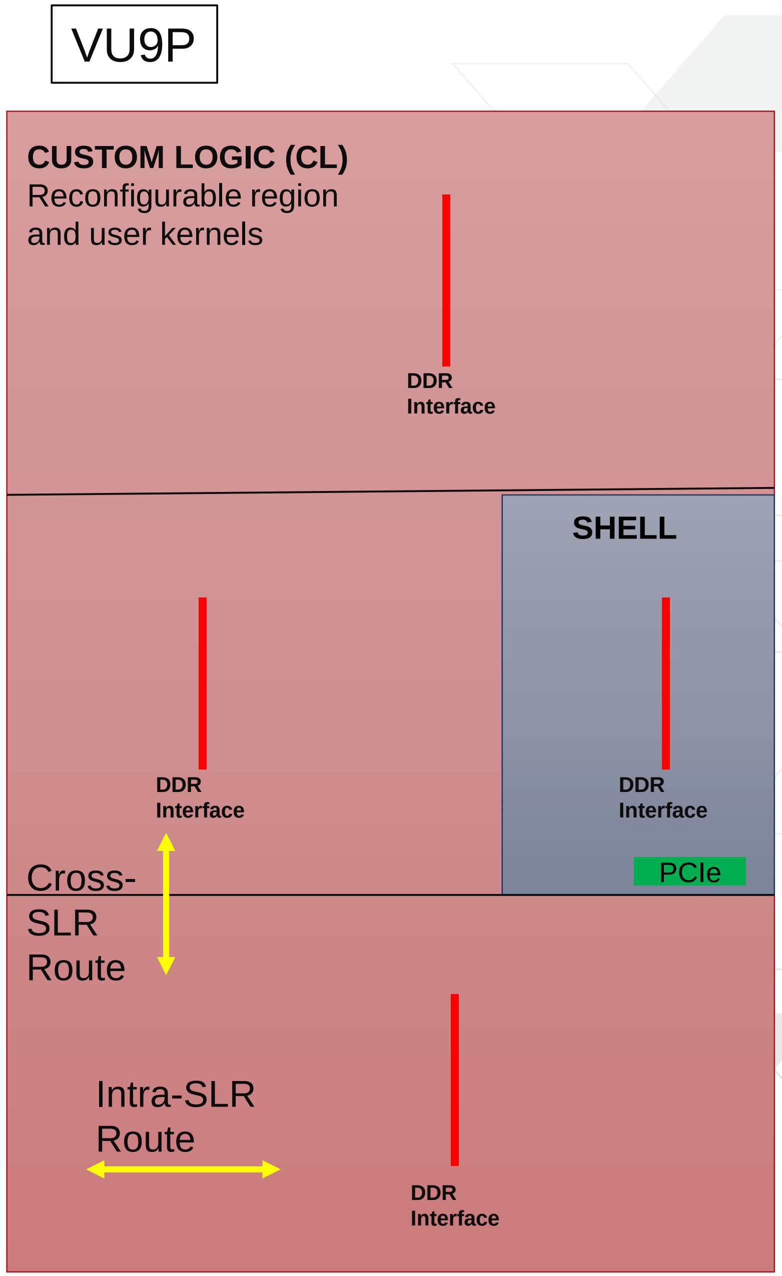
FPGA Physical View

> **Today's largest FPGA are stacked silicon devices with several SLRs (super logic regions)**

> **Connections between SLRs incurs a greater delay than standard intra-SLR routing**

> **By default, kernels are placed in the same SLR as the Shell**

> **Careful placement of kernels in SLRs will improve Fmax**
>> Aim to place kernels in the same SLR as the DDR they interface with
>> Aim to minimize SLR congestion and cross-SLR connections



© Copyright 2018 Xilinx

Understand the target DSA

> **Review Documentation**
  - SDAccel Release Notes (UG 1238)
    - DSA Released 2017.4
  - DSA Specifications

> **Shell**
  - Consumes Resources to implement
  - Removes available resources from the dyna
  - Understand which SLRs are impacted

SLR resources and DDR assignment can impact performance

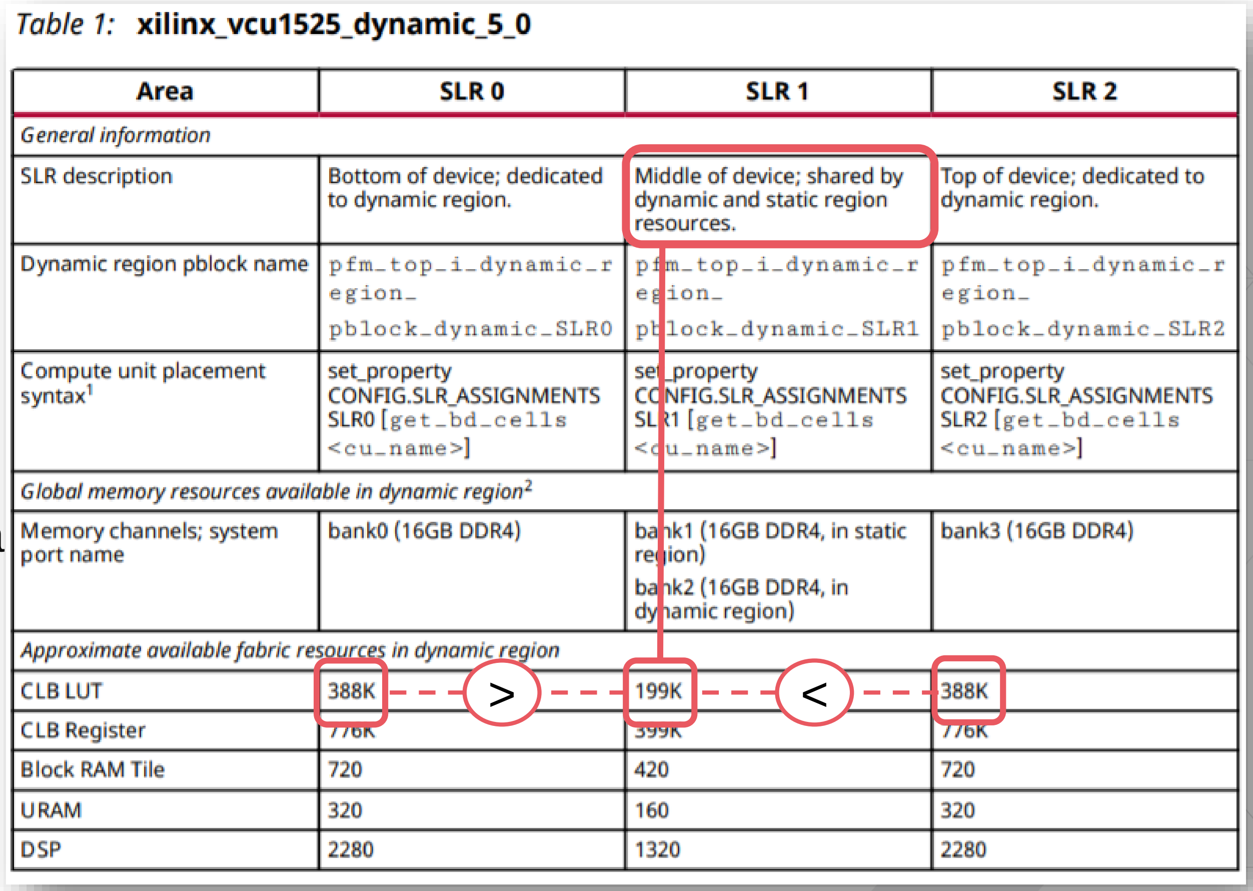


Table 1: **xilinx_vcu1525_dynamic_5_0**

Area	SLR 0	SLR 1	SLR 2
General information			
SLR description	Bottom of device; dedicated to dynamic region.	Middle of device; shared by dynamic and static region resources.	Top of device; dedicated to dynamic region.
Dynamic region pblock name	`pfm_top_i_dynamic_region_` `pblock_dynamic_SLR0`	`pfm_top_i_dynamic_region_` `pblock_dynamic_SLR1`	`pfm_top_i_dynamic_region_` `pblock_dynamic_SLR2`
Compute unit placement syntax[1]	set_property CONFIG.SLR_ASSIGNMENTS SLR0 `[get_bd_cells <cu_name>]`	set_property CONFIG.SLR_ASSIGNMENTS SLR1 `[get_bd_cells <cu_name>]`	set_property CONFIG.SLR_ASSIGNMENTS SLR2 `[get_bd_cells <cu_name>]`
Global memory resources available in dynamic region[2]			
Memory channels; system port name	bank0 (16GB DDR4)	bank1 (16GB DDR4, in static region) bank2 (16GB DDR4, in dynamic region)	bank3 (16GB DDR4)
Approximate available fabric resources in dynamic region			
CLB LUT	388K	199K	388K
CLB Register	776K	399K	776K
Block RAM Tile	720	420	720
URAM	320	160	320
DSP	2280	1320	2280

$388K > 199K < 388K$

© Copyright 2018 Xilinx

Kernel Placement Control

> **Specify Kernel Locations**

>> Provided by XOCC command line arguments

```
xocc <arguments> --xp param:compiler.userPostSysLinkTcl=<path>/place_krnl.tcl
```

– Auto-executes a Vivado Tcl file

>> Kernel locations specified by Vivado Tcl script

place_krnl.tcl

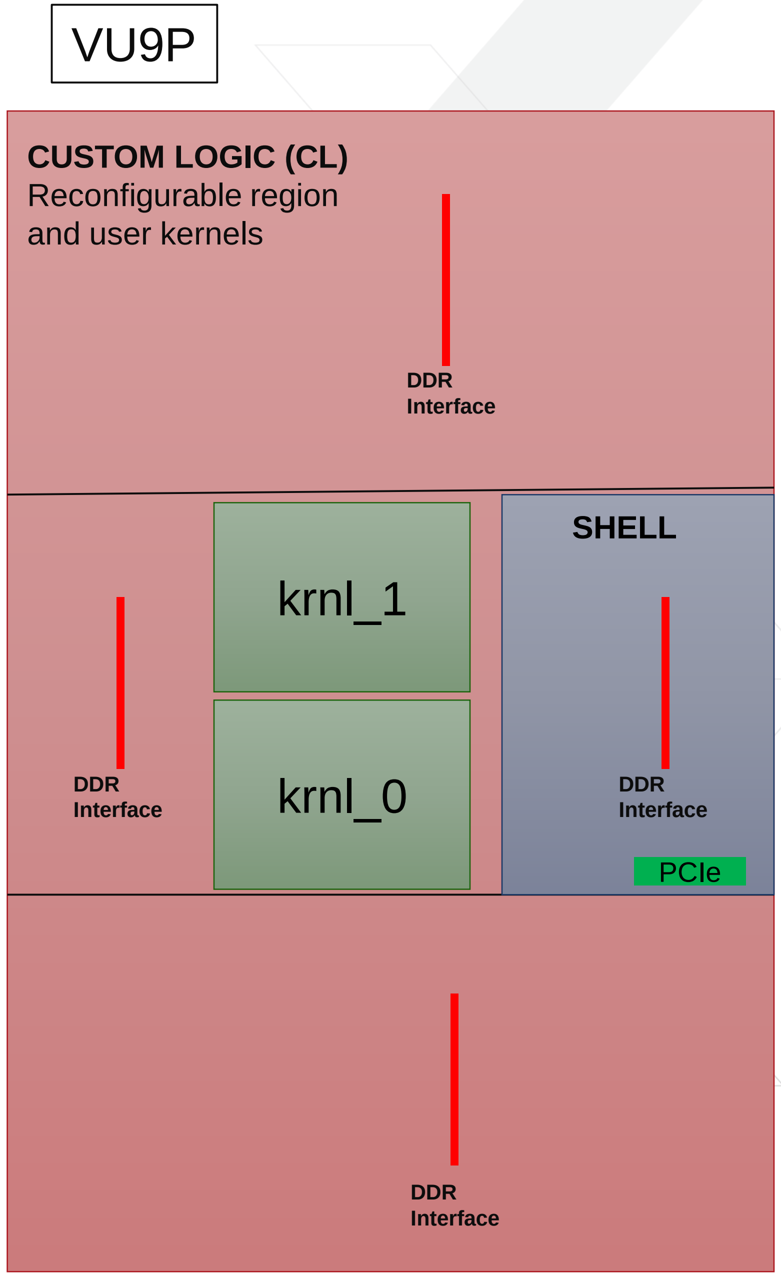
```
set_property CONFIG.SLR_ASSIGNMENTS SLR0 [get_bd_cells /krnl_0]
set_property CONFIG.SLR_ASSIGNMENTS SLR2 [get_bd_cells /krnl_1]
```

> **Command line option provided in 2018.3**

>> Requires a new DSA revision (5.2)

>> Also ensures local SLR reset is used

```
xocc <arguments> --slr  krnl_0:SLR0 --slr  krnl_1:SLR2
```



© Copyright 2018 Xilinx

Performance Profiling with SDAccel

© Copyright 2018 Xilinx

Main Report Files

- SDAccel generates important report files that help to improve performance

- Reports from Hardware Emulation are most useful for performance improvement
  - Guidance Report

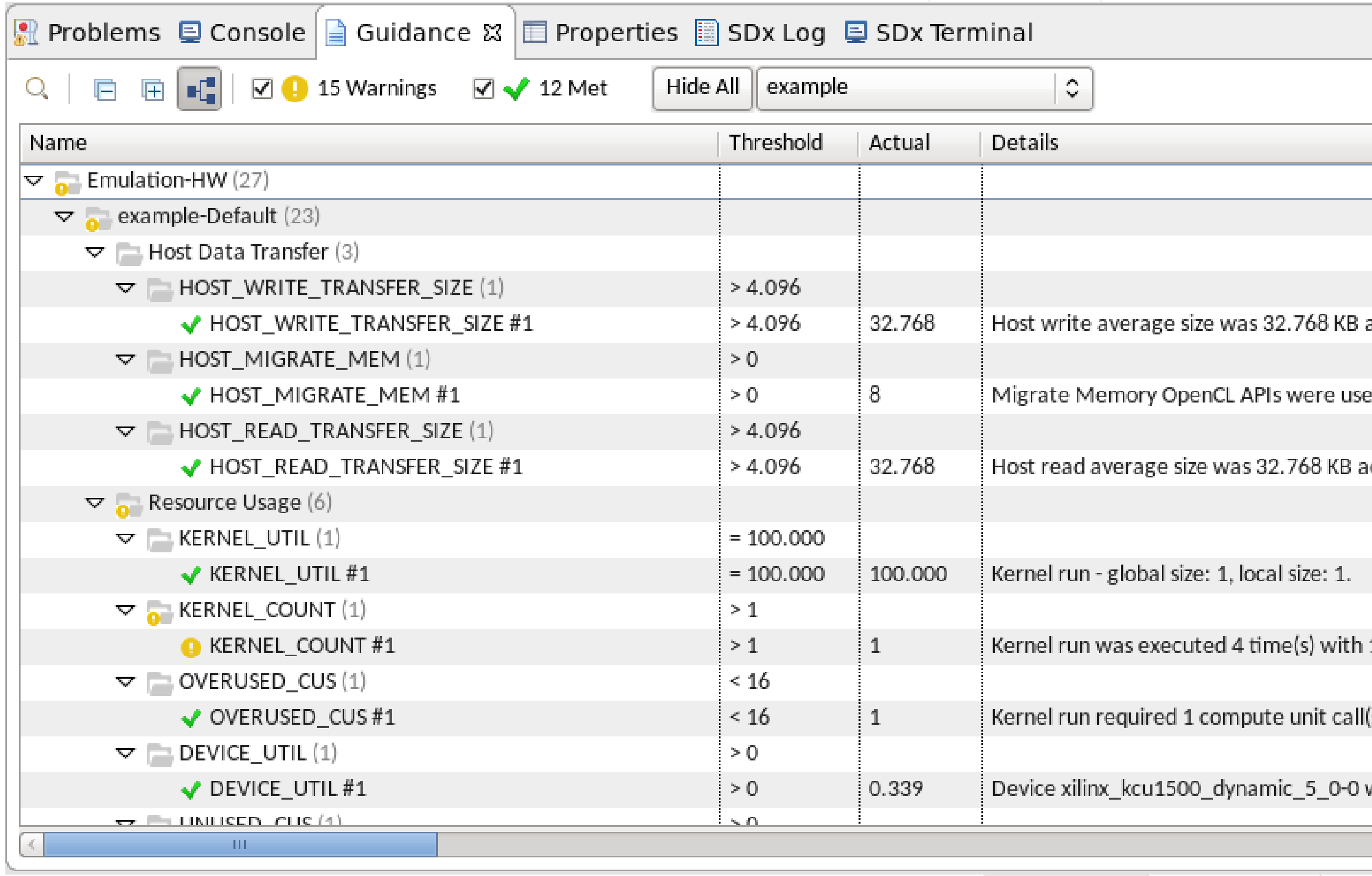
- Reports to analyze overall system performance (combining Host and Kernel)
  - Profile Summary Report
  - Timeline Trace
  - Waveform

- Reports to understand/improve Kernel performance
  - HLS report
  - Schedule Viewer

© Copyright 2018 Xilinx

Design Guidance

- **Expert system built-in the tool**
  - Analysis of build results and emulation runs
- **Guidance window with feedback by category**
  - Host code
  - Kernels
  - Data transfers (host to DDR, DDR to kernels)
- **Explicit and actionable hints**
  - How to improve the design
  - Links to detailed explanation and solutions
  - HTML report (available for makefile runs as well)

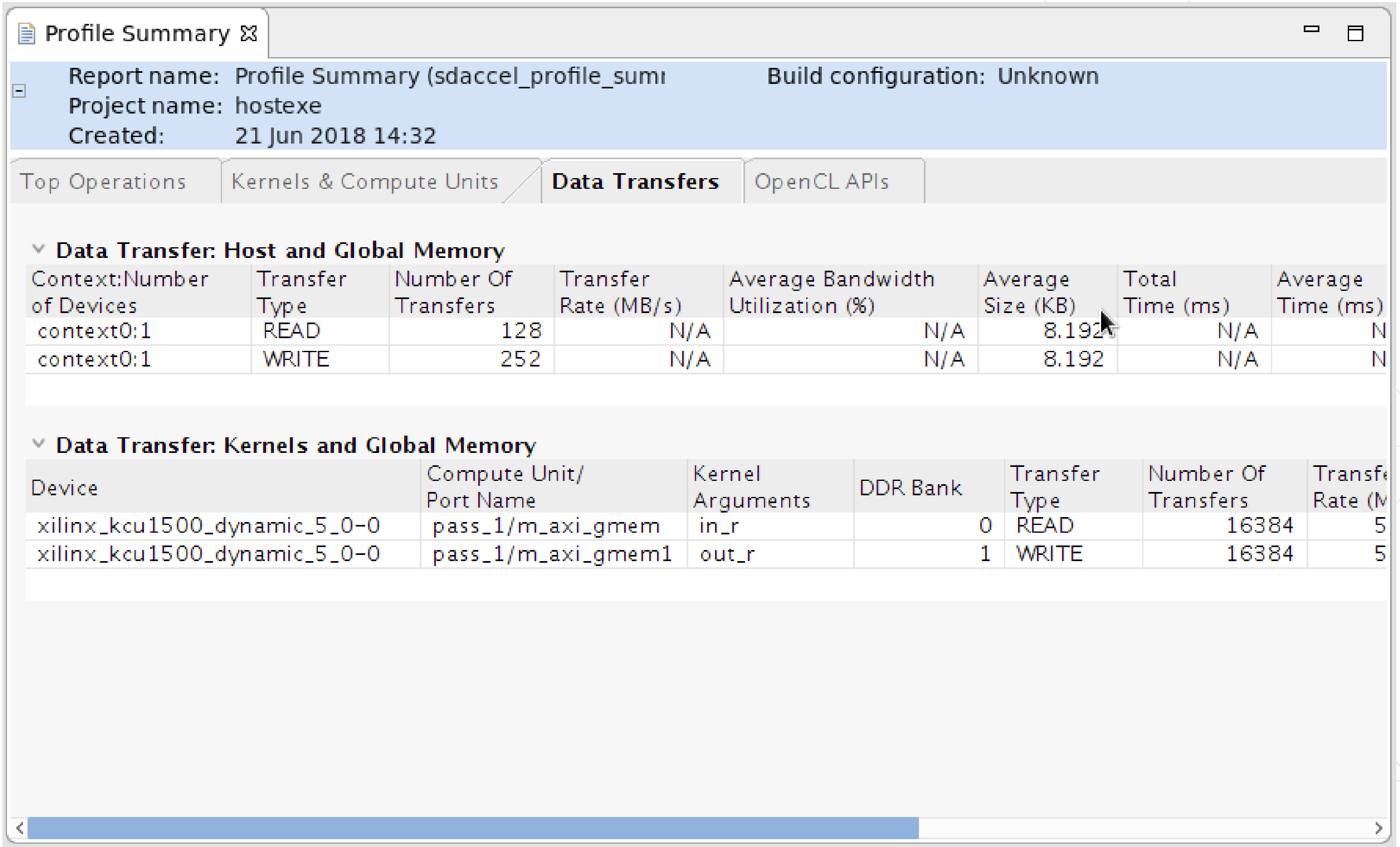


Name	Threshold	Actual	Details
Emulation-HW (27)			
example-Default (23)			
Host Data Transfer (3)			
HOST_WRITE_TRANSFER_SIZE (1)	> 4.096		
✓ HOST_WRITE_TRANSFER_SIZE #1	> 4.096	32.768	Host write average size was 32.768 KB a
HOST_MIGRATE_MEM (1)	> 0		
✓ HOST_MIGRATE_MEM #1	> 0	8	Migrate Memory OpenCL APIs were use
HOST_READ_TRANSFER_SIZE (1)	> 4.096		
✓ HOST_READ_TRANSFER_SIZE #1	> 4.096	32.768	Host read average size was 32.768 KB a
Resource Usage (6)			
KERNEL_UTIL (1)	= 100.000		
✓ KERNEL_UTIL #1	= 100.000	100.000	Kernel run - global size: 1, local size: 1.
KERNEL_COUNT (1)	> 1		
! KERNEL_COUNT #1	> 1	1	Kernel run was executed 4 time(s) with
OVERUSED_CUS (1)	< 16		
✓ OVERUSED_CUS #1	< 16	1	Kernel run required 1 compute unit call(
DEVICE_UTIL (1)	> 0		
✓ DEVICE_UTIL #1	> 0	0.339	Device xilinx_kcu1500_dynamic_5_0-0 v
UNUSED_CUS (1)	> 0		

© Copyright 2018 Xilinx

Profile Summary

- > **Top Operations**
  - >> Activity summary
- > **Kernel & Compute Units**
  - >> Detailed execution statistics
- > **Data Transfer**
  - >> Global Memory access statistics from host and from kernels
- > **Host Code OpenCL API statistics**



Report name: Profile Summary (sdaccel_profile_sumı   Build configuration: Unknown
Project name: hostexe
Created: 21 Jun 2018 14:32

Data Transfer: Host and Global Memory

Context:Number of Devices	Transfer Type	Number Of Transfers	Transfer Rate (MB/s)	Average Bandwidth Utilization (%)	Average Size (KB)	Total Time (ms)	Average Time (ms)
context0:1	READ	128	N/A	N/A	8.192	N/A	N
context0:1	WRITE	252	N/A	N/A	8.192	N/A	N

Data Transfer: Kernels and Global Memory

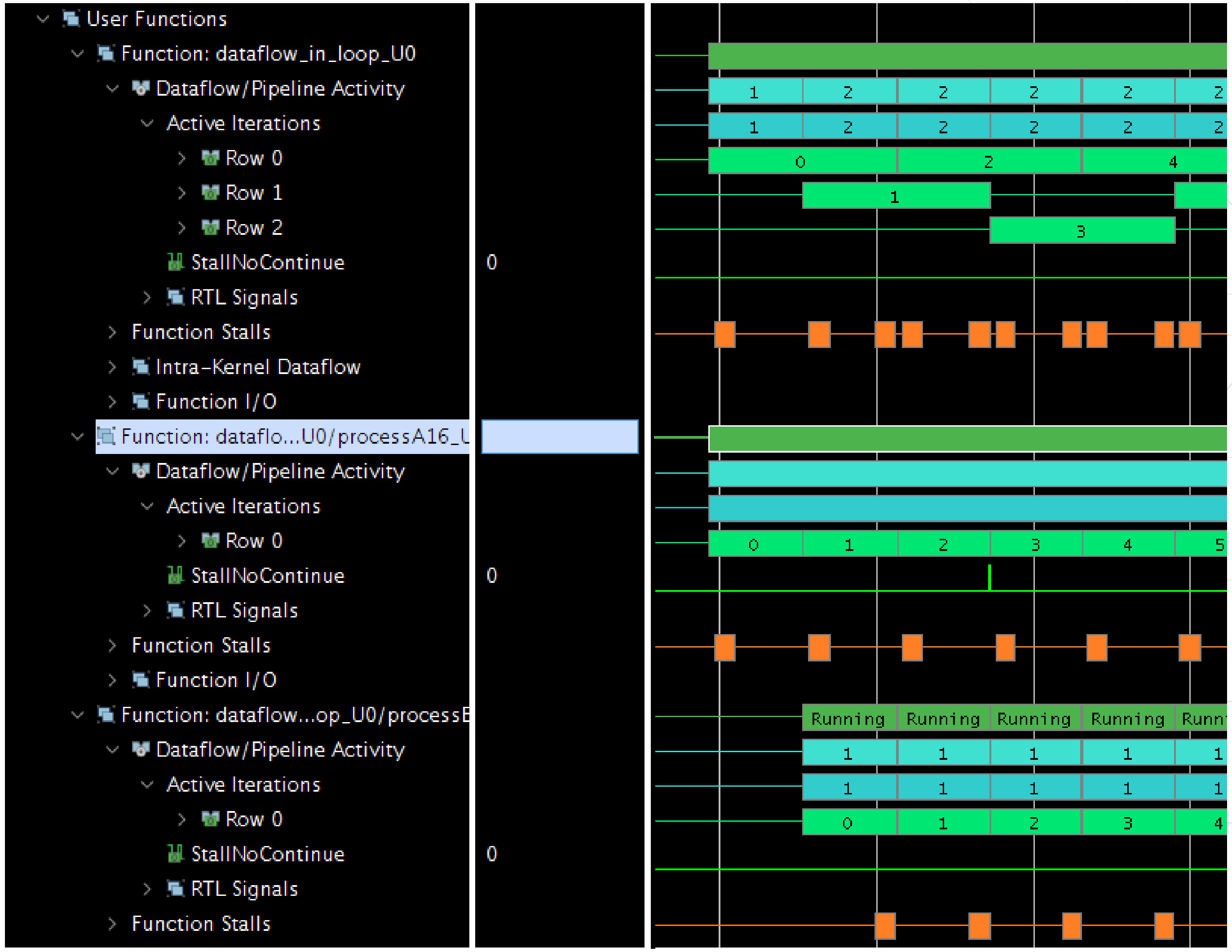
Device	Compute Unit/ Port Name	Kernel Arguments	DDR Bank	Transfer Type	Number Of Transfers	Transfe Rate (M
xilinx_kcu1500_dynamic_5_0-0	pass_1/m_axi_gmem	in_r	0	READ	16384	5
xilinx_kcu1500_dynamic_5_0-0	pass_1/m_axi_gmem1	out_r	1	WRITE	16384	5

© Copyright 2018 Xilinx

Annotated Waveform Viewer

> Shows task-level parallelism in action

> Show how many tasks overlap and for how long

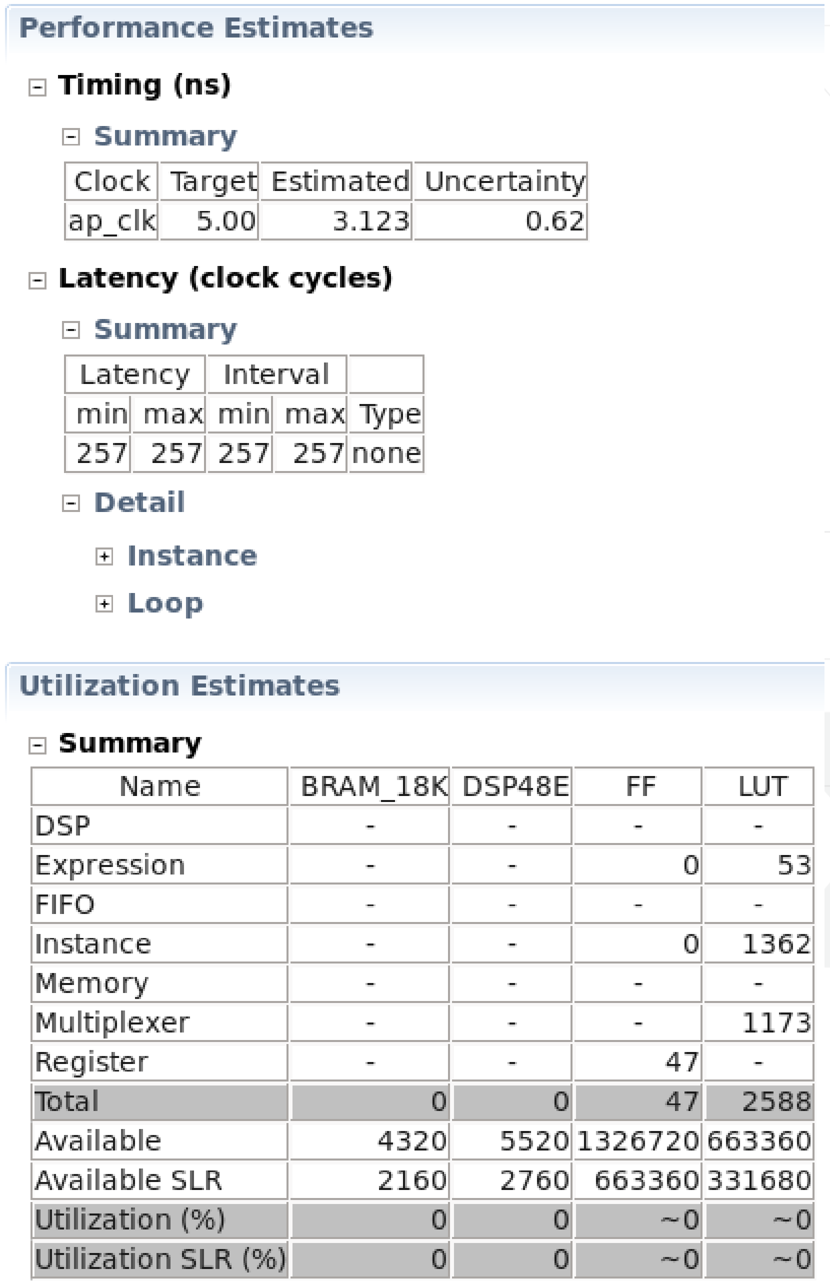


© Copyright 2018 Xilinx

HLS Report

> **Static Performance Estimates:**
  - >> Timing
  - >> Latency
  - >> Hierarchical contribution

> **Utilization Estimates:**
  - >> Summary
  - >> Detail analysis



Performance Estimates

⊟ **Timing (ns)**

⊟ **Summary**

Clock	Target	Estimated	Uncertainty
ap_clk	5.00	3.123	0.62

⊟ **Latency (clock cycles)**

⊟ **Summary**

Latency		Interval		
min	max	min	max	Type
257	257	257	257	none

⊟ **Detail**

⊞ **Instance**

⊞ **Loop**

Utilization Estimates

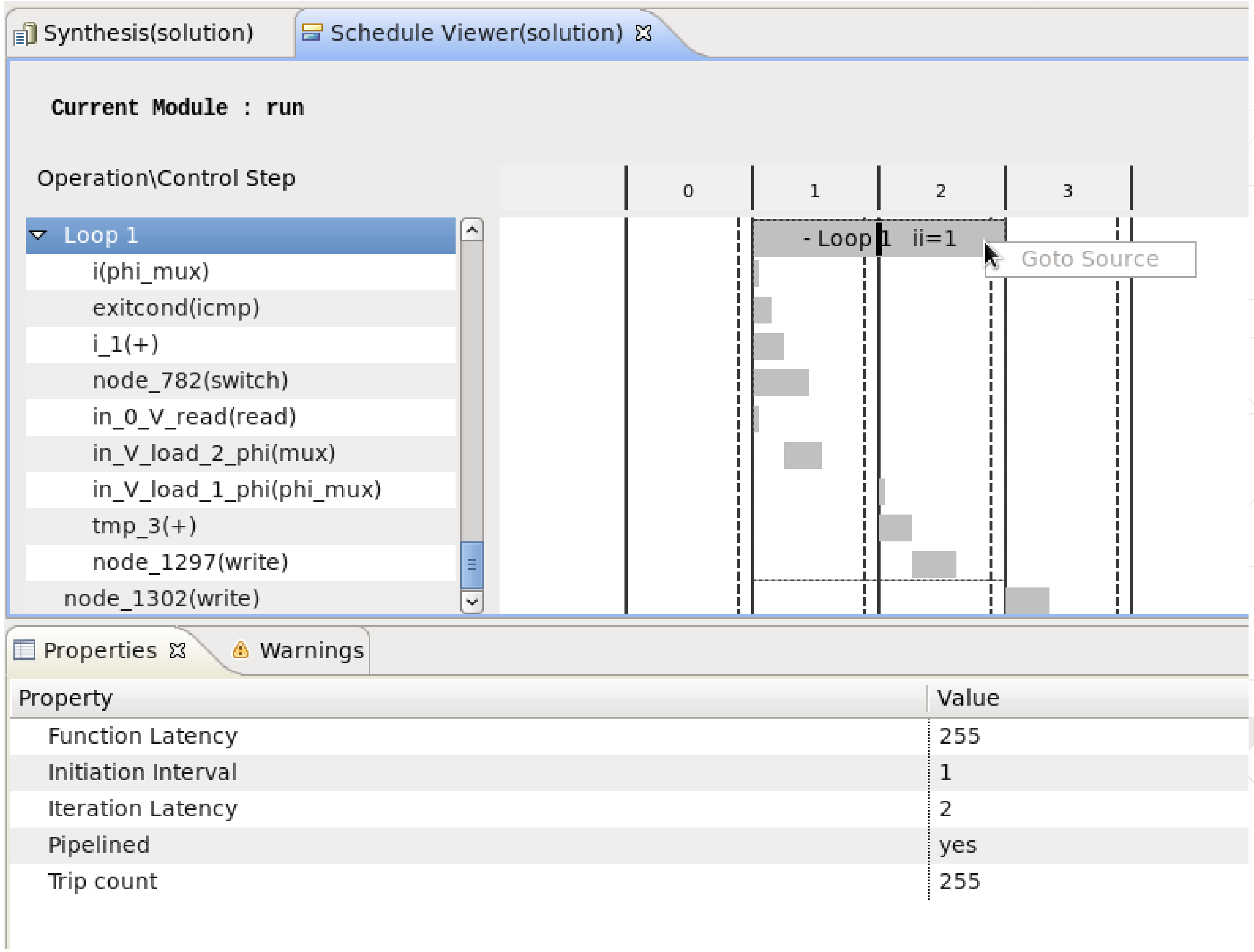
⊟ **Summary**

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	-	0	53
FIFO	-	-	-	-
Instance	-	-	0	1362
Memory	-	-	-	-
Multiplexer	-	-	-	1173
Register	-	-	47	-
Total	0	0	47	2588
Available	4320	5520	1326720	663360
Available SLR	2160	2760	663360	331680
Utilization (%)	0	0	~0	~0
Utilization SLR (%)	0	0	~0	~0

© Copyright 2018 Xilinx

HLS Schedule Viewer

- Shows in which cycle operations are scheduled
- Shows operator timing and clock margin
- Shows data dependencies
- Cross-probing from operations to source code
- Supports specific focus on:
  - II Violation
  - Timing Violation



Current Module : run

- Loop 1
  - i(phi_mux)
  - exitcond(icmp)
  - i_1(+)
  - node_782(switch)
  - in_0_V_read(read)
  - in_V_load_2_phi(mux)
  - in_V_load_1_phi(phi_mux)
  - tmp_3(+)
  - node_1297(write)
- node_1302(write)

Property	Value
Function Latency	255
Initiation Interval	1
Iteration Latency	2
Pipelined	yes
Trip count	255

© Copyright 2018 Xilinx

Summary

> **First address Guidance Suggestions provided by SDAccel**

> **Use performance analysis viewers to identify further optimization opportunities**

> **Consider all areas for Performance Optimization**
  - Host program optimizations
  - Kernel Code optimizations
  - Topological optimizations
  - Implementation optimizations

© Copyright 2018 Xilinx

More Details

> **UG1207: SDAccel Environment Optimization Guide**

> **SDAccel Examples:**
  >> https://github.com/Xilinx/SDAccel_Examples

© Copyright 2018 Xilinx

XDF XILINX DEVELOPER FORUM

© Copyright 2018 Xilinx