Optimization Techniques

SDx 2018.2

XILINX.

Objectives

- After completing this module, you will be able to:
  - Communicate optimization parameters to the compiler
  - Identify where the optimization can be made
  - List various optimization techniques

XILINX.

Outline

> Introduction
> Memory Access Optimization
> Data Path Optimization
> Summary
> Lab Intro

XILINX.

UG1207: SDAccel Optimization Guide

- **The guide covers**
  - General Recommendations
  - Host Optimization
  - Memory Access optimization
  - Data Path optimization
  - General Optimization Strategy & Performance Checklist

XILINX.

OpenCL and SDAccel Attributes

- Attributes - programmer hints to the OpenCL™ compiler
  - For performance optimization
- It is up to each compiler to follow or ignore them
- OpenCL supports various attributes
  - opencl_unroll_hint
  - reqd_work_group_size
  - …

```
__kernel void vmult(global int* a, global int* b, global int* c)
{
  int tid = get_global_id(0);

  __attribute__((opencl_unroll_hint(2)))
  for (int i=0; i<4; i++) {
      int idx = tid*4 + i;
      a[idx] = b[idx] * c[idx];
  }
}
```

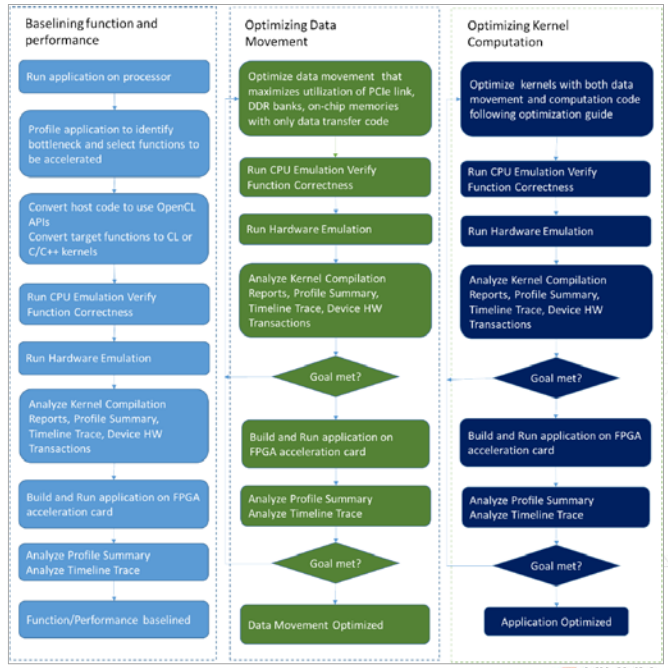

OpenCL and SDAccel Attributes

- SDAccel has specific FPGA optimization attributes
  - Not part of the OpenCL standard
  - ALL Xilinx attributes starts with "xcl" prefix:
    - xcl_pipeline_loop
    - xcl_pipeline_workitems
    - xcl_array_partition
- Use __xilinx__ macro to conditionally include SDAccel specific attributes in a kernel

```
#ifdef __xilinx__
  __attribute__((xcl_pipeline_loop))
#endif
for (int i=0; i<4; i++) {
    int idx = tid*4 + i;
    a[idx] = b[idx] * c[idx];
}
```

Optimization Strategy

> **Three Phases**
  >> Performance **Baselining**
  >> **Data Movement** Optimization
  >> **Kernel Computation** optimization



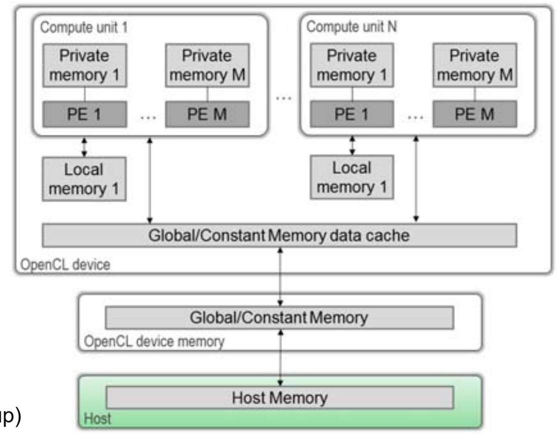
XILINX.

Memory Access Optimization

XILINX.

OpenCL: Five Sub-Regions of Memory Objects

- **Host Memory**
  - Visible to Host only
  - OpenCL ONLY defines how Host Memory interacts with OpenCL objects
- **Global Memory**
  - Visible to Host and Device
  - All Work Items in All Workgroups can read/write there
  - **Global on-chip Memory** – visible to Device only
- **Constant Memory**
  - Region of a Global memory
  - Work items – reads access only
- **Local Memory**
  - Local to a workgroup (shared by All work-items in a group)
- **Private Memory**
  - Accessible by a work-item



Memory Transfer

- **Optimization techniques considerations**
  - Using Multiple Memory Ports
    - Default is to use single global memory port to a kernel
    - Use multiple global memory ports through code
    - Use SDAccel option check box
  - Increase Port Width
    - SDAccel determines port width by analyzing kernel arguments
    - Use vector data types for a wide data path within the kernel
      - int16, int8, int4, int3, int2 (int 32 bits)
      - char16, char8, char4, char3, char2 (char 8 bits)
      - float16, float8, float4, float3, float2 (float 32 bits)

Memory Transfer

> **Optimization techniques considerations**
- Using Multiple Memory Ports
- Increase Port Width
- Using Local Memory + Burst data transfer
- Using On-Chip Global Memory

XILINX.

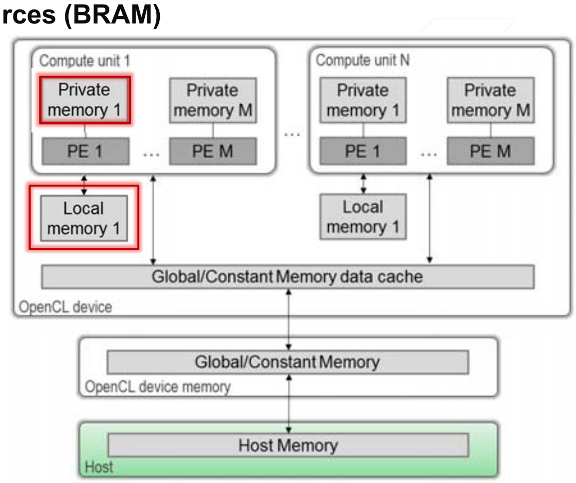
Using Local + Burst Data Transfer

> **Local Memory – implemented on FPGA resources (BRAM)**
- Lower latency and higher throughput
- Accessible by compute unit (within a workgroup)

> **Note: you can use private memory as well**

> **Move repeatedly used data to Local memory**
- Cache data - reduce redundant global memory access
- Improves global memory access patterns
- *Note: Check available BRAM resources*

> **To burst data, Global -> Local memory use**
- Pipelined Loops – recommended
- `async_work_group_copy` – not recommended



XILINX.

Using Local Memory + Burst data transfer

> **Example – Original Design: Data read from / written to Global memory**

```
__kernel
void K_VADD(__global int* A, __global int* B, __global int* R)
{
  int A_l=0, A_r=0;

  for (int i=0; i<MAX_Nb_Of_Elements; i++) {
    if(i==0) {
        A_l = 0; A_r = A[i+1];
    } else {
        if (i==(MAX_Nb_Of_Elements-1)) {
            A_l = A[i-1]; A_r = 0;
        } else {
            A_l = A[i-1]; A_r = A[i+1];
        }
    }
    R[i] = A[i] + (A_r - A_l) * B[i];
  }
}
```

Kernel Execution (includes estimat		
Kernel	Number Of Enqueues	Total Time (ms)
K_VADD	1	0.051

Number Of Transfers	Average Size (KB)
4094	0.004
1024	0.004

> **Modified Design: Data read from/written to local memory**

```
__kernel
void K_VADD(__global int* A, __global int* B, __global int* R)
{
  int A_l=0, A_r=0;

  __local int A_loc[MAX_Nb_Of_Elements], B_loc[MAX_Nb_Of_Elements], R_loc[MAX_Nb_Of_Elements];
  event_t events[2];

   __attribute__ ((xcl_pipeline_loop)) for (int k=0; k<MAX_Nb_Of_Elements; k++) A_local[k] = A[k];
   __attribute__ ((xcl_pipeline_loop)) for (int k=0; k<MAX_Nb_Of_Elements; k++) B_local[k] = B[k];

  for (int i=0; i<MAX_Nb_Of_Elements; i++) {
    ...
    R_loc[i] = A_loc[i] + (A_r - A_l) * B_loc[i];
  }

  __attribute__ ((xcl_pipeline_loop)) for (int k=0; k<MAX_Nb_Of_Elements; k++) R[k] = R_local[k];
}
```

Kernel Execution (includes estimat		
Kernel	Number Of Enqueues	Total Time (ms)
K_VADD	1	0.026

Number Of Transfers	Average Size (KB)
128	0.064
64	0.064

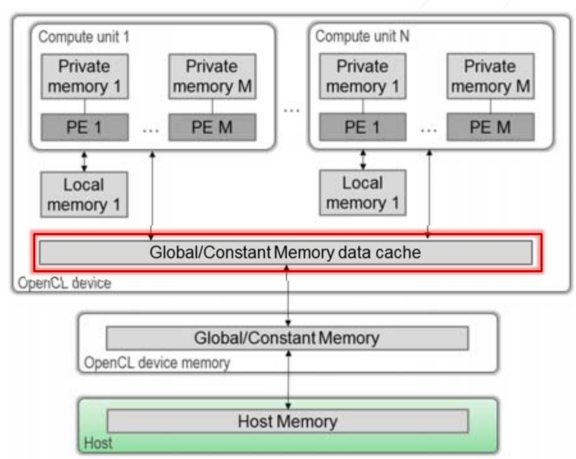
XILINX.

Using On-Chip Global Memory

> **In OpenCL 2.0 Specification**
>> Enables you to move buffers to FPGA
>> Can be used for intra-kernel communication
>> Not visible to Host

> **Implementation**
>> Statically allocated in BRAM at kernel compile time

> **Consideration**
>> Must be at least 4096 bytes



XILINX.

Memory Transfer

> **Optimization techniques considerations**
- Using Multiple Memory Ports
- Increase Port Width
- Using Local Memory + Burst data transfer
- Using On-Chip Global Memory
- Using Pipes
- Memory Partitioning

Using Pipes

- **Enables kernels to run in Parallel**
  - Defined in OpenCL 2.0 specification
- **FIFO storage for streaming data between kernels**
- **Implemented on the FPGA as FIFO**
  - Defined at kernel compile time
  - Cannot use `clCreatePipe` API
- **A pipe can only have ONE producer and ONE consumer across kernels**
- **Pipe Functions:**
  - read_pipe , write_pipe - built-in non-blocking OpenCL functions
  - read_pipe_block, write_pipe_block – Xilinx extension - blocking mode

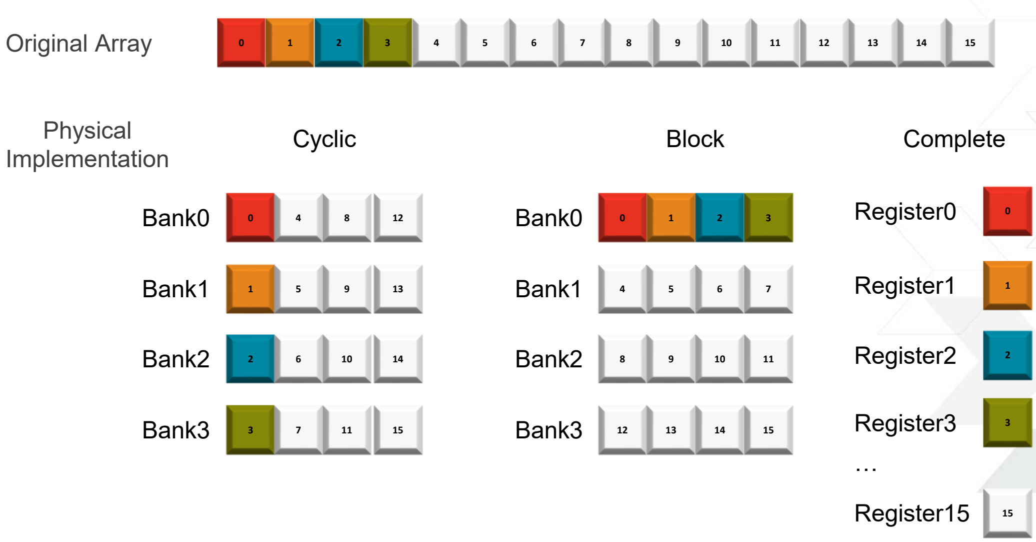
```
// Define a pipe of 16 - 32768 elements in powers of two: 2^N (4 <= N <= 15)
pipe int p0 __attribute__((xcl_reqd_pipe_depth(int)));

// Reading and writing from a pipe
int read_pipe (pipe gentype p, gentype *ptr)
int write_pipe (pipe gentype p, const gentype *ptr)
```

Memory Partitioning

- **Local, Private, Global On-Chip Memory - <u>usually BRAM implementation</u>**
  - <u>Recall</u>: BRAM - a dual-port RAM module
- **Access to BRAM can often be a <u>performance bottleneck</u>**
- **Array partitioning – modifies how data is stored in memory**
  - Implements an array as multiple physical memories (instead of single)
  - Improves performance
- **Partitioning can be: Block, Cyclic, Complete**
  - Depends on the application algorithm
- **Attribute:**

```
__attribute__((xcl_array_partition(<partition type>,
                                   <partition factor>,
                                   <array dimension>)))
```


Memory Partitioning Schemes



Data Path Optimization

XILINX.

Datapath Optimization

> **Optimization Techniques**
- Loop Pipelining
- Loop Unrolling
- Dataflow

XILINX.

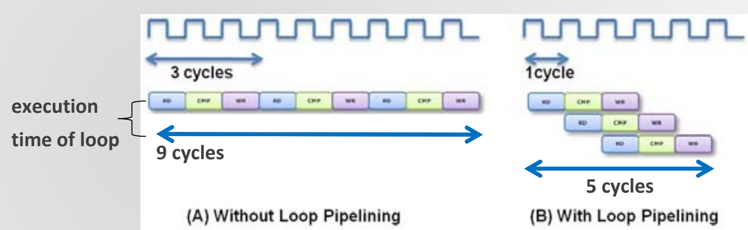
HLS Optimization Pragmas

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 Pipelining

> **Pipelining - keeps all kernel logic elements busy at all times.**

```
kernel void
foo(...)
{
  __attribute__((xcl_pipeline_loop))
  for (int i=0; i<3; i++) {
    int idx = get_global_id(0)*3 + i;
    op_Read(idx);
    op_Compute(idx);
    op_Write(idx);
  }
}
```



> **Attribute:**

`__attribute__((xcl_pipeline_loop))`

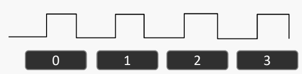
> **SDAccel pipelines loops automatically**

>> Use HLS report to see if loops are pipelined

Loops – Latency

- **Loop iteration runs on the same HW resources**
  - e.g. an accumulation in a loop is one adder
- **Loops imply latency**
- **Incrementing a loop counter always consumes 1 clock cycle**
  - *(at least by default and in the absence of directives)*

```
void foo (...)  {
...
add: for (i=0;i<=3;i++)
{
  b = a[i] + b;
...
```

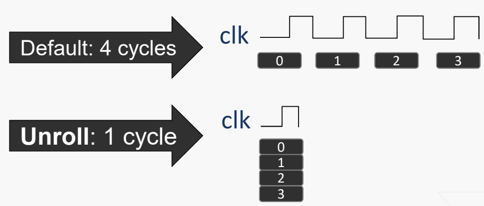


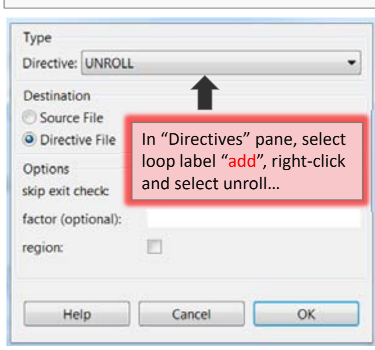
Example: This loop (without directives) will always take at least 4 clock cycles

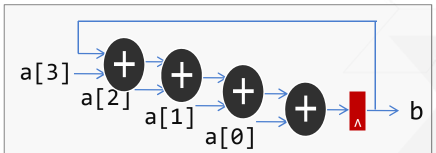
XILINX.

Loops – Unrolling

```
void foo (...)  {
...
add: for (i=0;i<=3;i++)
{
  b = a[i] + b;
...
```







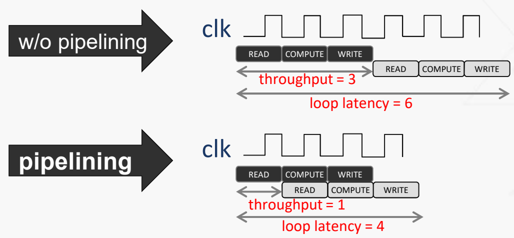
Example: Fully unrolled loop.
(parallel execution but more area)

XILINX.

Loops – Pipelining

> **Pipelining allows for loop iterations to run in parallel**
>> Improves throughput (a.k.a initiation interval also referred to as II)

```
void foo (...)  {
...
add: for (i=1;i=<2;i++)
{
  op_READ;
  op_COMPUTE;
  op_WRITE;
}
...
```

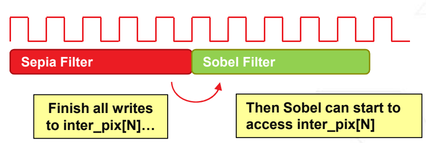


XILINX.

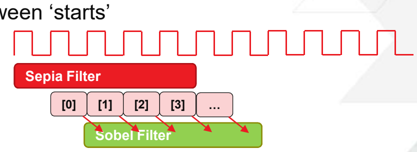
Dataflow

> **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)



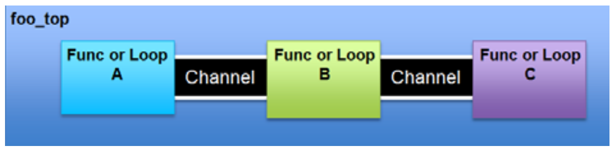
> **Apply dataflow within a function but not at the top level**

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

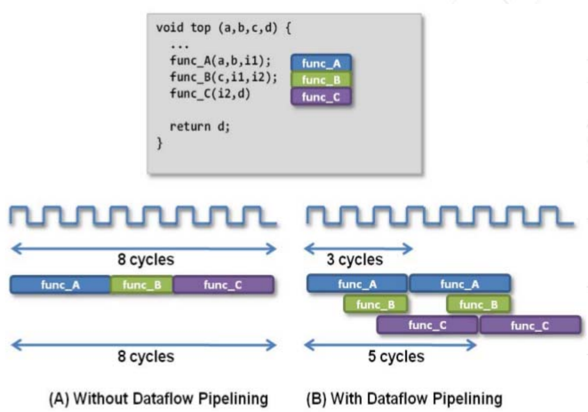
XILINX.

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.

Summary

Summary

- UG1207 describes optimization techniques
- OpenCL supports various attributes to optimize application
- Optimization parameters are communicated through __attribute__((<attribute>))
- Xilinx specific optimization can be described using __xilinx__ macro to conditionally include SDAccel specific attributes in a kernel
- Optimization techniques include
  - Host Optimization
  - Memory Access optimization
  - Data Path optimization

Lab Intro

Lab Intro

> In this lab you will apply DATAFLOW optimization technique to the kernel code and PIPELINE optimization technique to the host code. You will analyze profiling and timing reports of the HW emulation to understand the throughput and data transfer improvements

> You will enable `Use waveform for kernel debugging` option in SDAccel Run Configuration and analyze hdl simulator output

Adaptable.
Intelligent.

XILINX