Practice Using Channels with OpenCL™ on Intel® FPGAs

Exercise Manual

Software Requirements that cannot be adjusted:
Intel FPGA SDK for OpenCL version 17.1
g++ compiler installed

Software Requirements that can be adjusted:
Operation System: CentOS 6.9 (Linux)

Use the link below to download the design files for the exercise:

https://www.altera.com/customertraining/OLT/OpenCLChannels/lab.zip

In this exercise, you will take an existing kernel and convert it to 3 kernels separating the process of reading and writing from the device global memory from the actual execution of the data processing. You will create channels to handle the kernel-to-kernel communications and then run the three kernels concurrently using three command queues.

In the original kernel file, there is only one active kernel, process_data which reads and writes from global memory and performs a simple multiplication on the original floating point number. After the exercise we should have three kernels, one to read from global memory, one to perform the exponential function, and one to write to global memory. We will run the kernel in AOC Emulation mode.

The solutions are in the Solutions/ subdirectory of the lab file folder.

Part 1. Modify a kernel to use channels

____ 1. Unzip the lab files using the command **unzip lab.zip**

____ 2. Change to the lab/ directory by typing **cd lab**

____ 3. Open the file in opencl_init.sh and change the paths to paths in your system to the Intel FPGA SDK for OpenCL installation directory

____ 4. Set up your environment for the lab by typing **source opencl_init.sh**

____ 5. In the terminal, compile the kernel we will be converting to channels into an emulation file by running the following command

```
aoc -march=emulator -board=a10gx not_channels.cl
```

____ 6. Build the host application by typing **source simple_compile.sh**. This compiles the main.cpp code using g++.

____ 7. Run the host application, which calls the emulated kernel by typing **./SimpleOpenCLApp**. You should see a message that says VERIFICATION PASSED! along with some sample results.

____ 8. We will now create the same functionality present in the not_channels.cl file in 3 separate kernels which use channels to pass data to one another.

   a. Open **channels.cl** in a text editor.

   b. Add the **pragma** that enables the extension for channels on the top line.

   c. Create two channels of type float and name them **c0** and **c1**, and set their depth to 128.

   Remember that channels are file scope variables.

____ 9. Examine the **process_data** kernel and notice there are no longer any buffers of data being passed by the host, so its only argument is a scalar.

____ 10. For the other two kernels, since we don't have any pointer aliasing add the **restrict** keyword after the * to the arguments of those kernels.

The restrict keyword allows the aoc compiler to treat each pointer as separate content and enables dependency optimizations.

____ 11. Make all of the kernels single work-item kernels.

  a. Add the __attribute__((max_global_work_dim(0))) to all of the kernels.

____ 12. For the **host_reader** kernel, add the necessary argument and functionality. This kernel takes in a buffer from the host and writes the data to a channel for processing.

  a. Add a 2nd argument **unsigned N** which allows the host to set the number of iterations to run

  b. Run a loop with N iterations, every iteration read 1 element from the input array and use the blocking call to write it to c0

Loops in single work-item kernels are pipelined, so we can expect to write to the channel roughly once per clock cycle.

____ 13. Examine the changes made to the **process_data** kernel.

  a. There is now only one kernel argument, which corresponds to the number of calculations to perform.

  b. Instead of reading from an input array, a channel is read within the loop to get the data to calculate upon.

  c. Instead of writing to an output array, a channel is written within the loop to pass the data to the host_writer kernel.

____ 14. For the **host_writer** kernel, add the extra arguments and functionality.

  a. Create a 2nd argument **unsigned N** which allows the host to pass in the number of iterations to run

  b. Run a loop with N iterations, every iteration read 1 element from channel c1 and assigned it to **value**.

  c. Then assign **value** to the output array.

____ 15. Save the channels.cl file.

____ 16. Go to the **terminal.** Ensure you are in the Ex3/ directory. Compile the kernel you just wrote for emulation using the following command:

```
aoc -march=emulator -board=a10gx channels.cl
```

Part 2. Modify the host code to run the channels kernel

____ 1. Open **main.cpp** in a text editor.

____ 2. In **main.cpp,** look for the comment “TODO: Add command queues”

Use the cl::CommandQueue constructor to construct two additional command queues. Name them queue1 and queue3 (queue2 has already been created for you). Use the same context and device as queue2.

In this lab, queue1 will launch the host_reader kernel, queue2 will launch the process_data kernel and queue3 will launch the host_writer kernel.

____ 3. Look for the comment “TODO: adjust which queue data is written to”

Adjust the queue the buffer is written to. It should correspond to the queue for the host_reader kernel.

Also adjust the queue for the finish() method call to correspond to the queue for the host_reader kernel.

____ 4. Look for the comment “TODO: change the name of the .aocx file”

Change the name of the .aocx file to channels.cl since we have written a new kernel file.

____ 5. Look for the comment “TODO: Create new kernels”

Use the cl::Kernel constructor to construct kernel objects for the host_reader and host_writer kernels.

____ 6. Look for the comment “TODO: Set arguments for all kernel”

Set up all of the arguments for the kernels.

a. The **host_reader** kernel has two arguments to set up here. Buffer_in and vectorSize.

b. The **process_data** kernel has one arguments to set up here, vectorSize.

c. The **host_writer** kernel has two arguments to set up here, Buffer_out and vectorSize

____ 7. Look for the comment “TODO: launch additional kernels”

Launch the 2 kernels that were added to the .cl file. Use the correct corresponding queue objects (variables) for each kernel launch. They can all be launched as tasks.

____ 8. Look for the comment “TODO: Wait on all the queues”

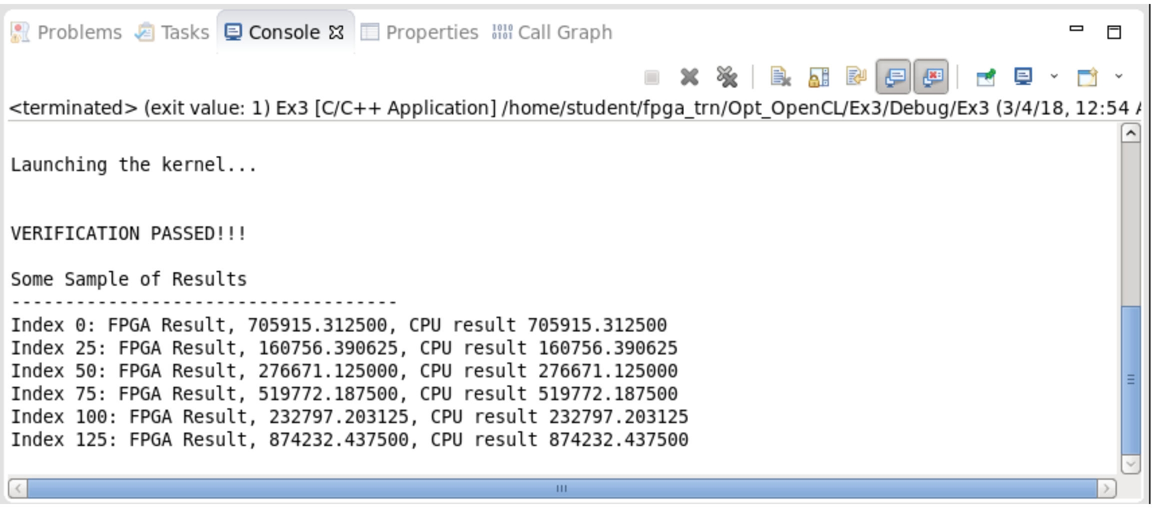
Since there are multiple queues executing kernels now, the flow control that an order queue imposes is not enough to keep the host code in sync with the kernel. Add a finish call for each queue.

____ 9. Look for the comment “TODO: Change the queue for reading data”

The queue the calculated data is being read from has changed. Change the queue for the enqueueReadBuffer and finish calls to reflect this.

____ 10. Save your code and compile the host code using the command **source simple_compile.sh**. Correct any compile errors.

____ 11. Run and debug the design. You should see a VERIFICATION PASSED! message similar to the below when your kernel successfully runs and calculates the correct results.



```
<terminated> (exit value: 1) Ex3 [C/C++ Application] /home/student/fpga_trn/Opt_OpenCL/Ex3/Debug/Ex3 (3/4/18, 12:54
```

```

Launching the kernel...


VERIFICATION PASSED!!!

Some Sample of Results
------------------------------------
Index 0: FPGA Result, 705915.312500, CPU result 705915.312500
Index 25: FPGA Result, 160756.390625, CPU result 160756.390625
Index 50: FPGA Result, 276671.125000, CPU result 276671.125000
Index 75: FPGA Result, 519772.187500, CPU result 519772.187500
Index 100: FPGA Result, 232797.203125, CPU result 232797.203125
Index 125: FPGA Result, 874232.437500, CPU result 874232.437500
```

Step 3. Examine the report files for the kernel

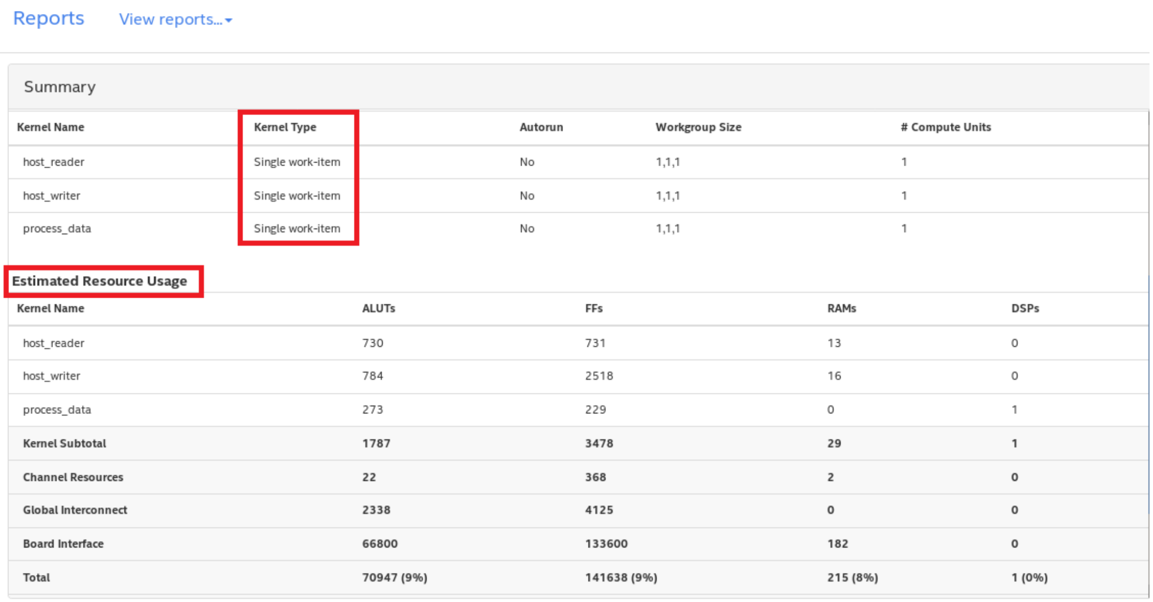
____ 1. In the terminal type

aoc -c -board=a10gx channels.cl

to compile the kernel and generate reports.

____ 2. Type **firefox channels/reports/report.html** to see the optimization report (or use your favorite web browser). Examine the summary section.

You can see that all of the kernels are compiled as single work-item kernels, and get an overview of the resource usage of the kernel.



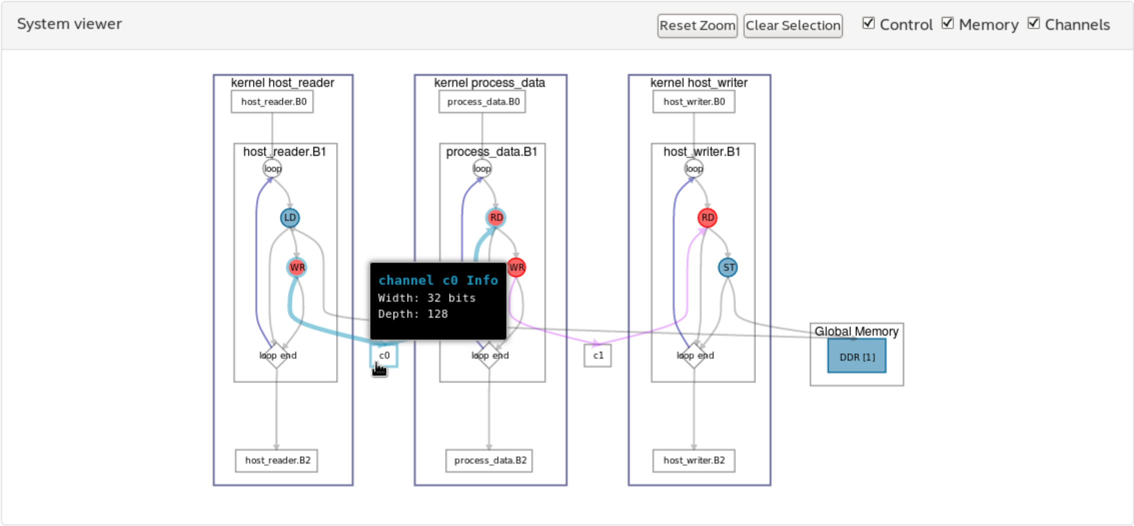
Summary

Kernel Name	Kernel Type	Autorun	Workgroup Size	# Compute Units
host_reader	Single work-item	No	1,1,1	1
host_writer	Single work-item	No	1,1,1	1
process_data	Single work-item	No	1,1,1	1

Estimated Resource Usage

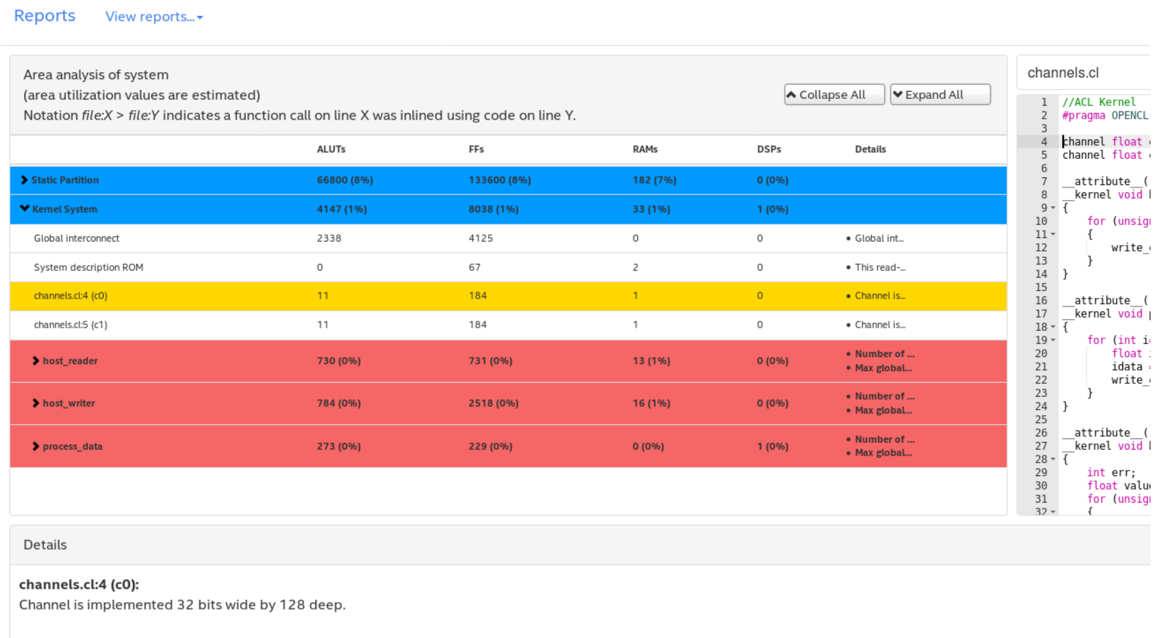
Kernel Name	ALUTs	FFs	RAMs	DSPs
host_reader	730	731	13	0
host_writer	784	2518	16	0
process_data	273	229	0	1
Kernel Subtotal	**1787**	**3478**	**29**	**1**
Channel Resources	**22**	**368**	**2**	**0**
Global Interconnect	**2338**	**4125**	**0**	**0**
Board Interface	**66800**	**133600**	**182**	**0**
Total	**70947 (9%)**	**141638 (9%)**	**215 (8%)**	**1 (0%)**

____ 3. Go to the System View section of the HTML report. Notice the channels within the System diagram, and hover over one for more information about it.



```
channel c0 Info
Width: 32 bits
Depth: 128
```

____ 4. Go to one of the Area Report sections of the HTML report. Notice where the resources for the channels appear and click on one to get more details.



Area analysis of system
(area utilization values are estimated)
Notation *file:X* > *file:Y* indicates a function call on line X was inlined using code on line Y.

	ALUTs	FFs	RAMs	DSPs	Details
Static Partition	**66800 (8%)**	**133600 (8%)**	**182 (7%)**	**0 (0%)**	
Kernel System	**4147 (1%)**	**8038 (1%)**	**33 (1%)**	**1 (0%)**	
Global interconnect	2338	4125	0	0	• Global int...
System description ROM	0	67	2	0	• This read-...
channels.cl:4 (c0)	11	184	1	0	• Channel is...
channels.cl:5 (c1)	11	184	1	0	• Channel is...
host_reader	**730 (0%)**	**731 (0%)**	**13 (1%)**	**0 (0%)**	**• Number of ...** **• Max global...**
host_writer	**784 (0%)**	**2518 (0%)**	**16 (1%)**	**0 (0%)**	**• Number of ...** **• Max global...**
process_data	**273 (0%)**	**229 (0%)**	**0 (0%)**	**1 (0%)**	**• Number of ...** **• Max global...**

channels.cl

```
//ACL Kernel
#pragma OPENCL

channel float
channel float

__attribute__(
__kernel void
{
    for (unsig
    {
        write_
    }
}

__attribute__(
__kernel void
{
    for (int i
        float
        idata
        write_
    }
}

__attribute__(
__kernel void
{
    int err;
    float valu
    for (unsig
    {
```

Details

channels.cl:4 (c0):
Channel is implemented 32 bits wide by 128 deep.

Exercise Summary

- Used a channel and a pipe to facilitate kernel to kernel communication
- Launched multiple kernels in parallel through the use of multiple queues