Application Acceleration with High-Level Synthesis

GitHub：https://github.com/ZheChen-Bill/convolution-filtering

lab_B 111061545 陳揚哲

Lab introduction:

In this lab, we use the acceleration card (u50) to accelerate the convolution filtering. We use a designed kernel runs on FPGA and discuss the optimization of the host-side application for performance. The kernel is designed to maximize throughput, and the host application is optimized to transfer data in effective methods that move in and between the host and FPGA. To eliminates the data movement latency, the data transfer must be overlapped for multiple kernel. Finally, we compare the estimate performance of hardware kernels using Vitis HLS and these estimates of actual hardware performance.

The lab is divided into 4 parts, which are 1. Accelerating Video Convolution Filtering Application, 2. Video Convolution Filter: Introduction and Performance Estimation, 3. Design and Analysis of Hardware Kernel Module for 2-D Video Convolution Filter, and 4. Building the 2-D Convolution Kernel and Host Application. Consequently, we will discuss the 4 parts respectively and summarize the lab.

0. Build the environment of Vitis-tutorials:

We clone the repository from github with following command.

```
git clone https://github.com/Xilinx/Vitis-Tutorials.git
```

We extract the large files in the directory under the convolution tutorials

```
cd /Vitis_Tutorials /Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial
wget https://www.xilinx.com/bin/public/openDownload?filename=conv_tutorial_files.tar.gz
-O conv_tutorial_files.tar.gz tar -xvzf    conv_tutorial_files.tar.gz
```

After downloading the files, we should set up the Vitis tools.

```
source <XILINX_VITIS_INSTALL_PATH>/settings64.sh
source <XRT_INSTALL_PATH>/setup.sh
```

Once the files and tools be set up, we can start to perform the serial labs.

1. Accelerating Video Convolution Filtering Application:

The first part is letting us the execute the acceleration by porting the video filter to acceleration card. That is, we directly execute hardware acceleration code, then compare the performance between baseline performance (CPU) and Hardware Acceleration.

First, we build the baseline of application performance. The software application processes the images with 1920*1080 resolution. Performing convolution on a set of images and prints the summary of performance results.

Software run is used for measuring baseline software performance. Run the application to measure performance as follows:

```
cd ./sw_run
./run.sh
```

We will get the summary of about the performance. (CPU)

```
------------------------------------------------------------
Number of runs     : 60
Image width        : 1920
Image height       : 1080
Filter type        : 6

Generating a random 1920x1080 input image
Running Software version on 60 images

CPU  Time          :     17.6637 s
CPU  Throughput    :     20.1519 MB/s
```

Then, running FPGA accelerated application to estimate the acceleration rate. The application will be run on an actual FPGA card, also called System Run. Run the following code the launch the FPGA accelerated video convolution filter.

```
cd /Vitis_Tutorials /Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial
make run
```

We will get the summary of about the performance. (hardware accelerated)

```
Xilinx 2D Filter Example Application (Randomized Input Version)

FPGA binary       : ./fpgabinary.hw.xclbin
Number of runs    : 60
Image width       : 1920
Image height      : 1080
Filter type       : 3
Max requests      : 6
Compare perf.     : 1

Programming FPGA device
XRT build version: 2.13.466
Build hash: f5505e402c2ca1ffe45eb6d3a9399b23a0dc8776
Build date: 2022-04-14 17:43:11
Git branch: 2022.1
PID: 59065
UID: 1059
[Sat Apr  1 19:34:52 2023 GMT]
HOST: HLS01
EXE: /mnt/HLSNAS/01.LcglIk/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial/build/host.exe
[XRT] WARNING: Trace Buffer size is too big. The maximum size of 4095M will be used.
[XRT] WARNING: Trace buffer size for 0th. TS2MM is too big for memory resource.  Using 268435456 instead.
Generating a random 1920x1080 input image
Running FPGA accelerator on 60 images
Running Software version
Comparing results

Test PASSED: Output matches reference

FPGA Time         :     0.4201 s
FPGA Throughput   :   847.3611 MB/s
CPU  Time         :    17.6540 s
CPU  Throughput   :    20.1630 MB/s
FPGA Speedup      :    42.0255 x
```

From the results, we can observe the hardware accelerated application has much better performance than baseline (CPU-only performance).

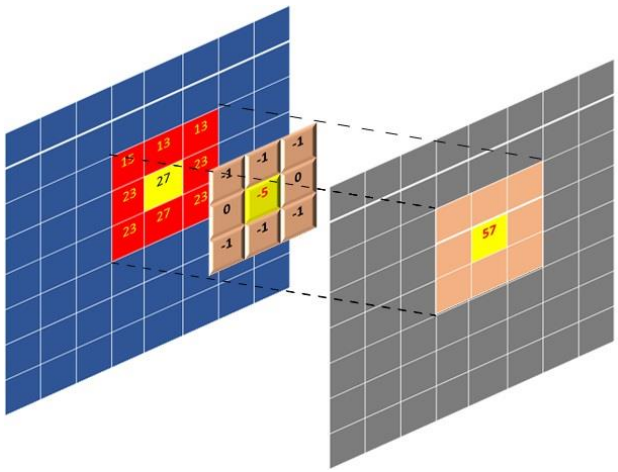
2. Video Convolution Filter: Introduction and Performance Estimation:

In second part, we explore the 2D video convolution filter and measure its performance on the host machine. The measurement is same as the baseline in first part.

We will learn the video convolution filter, measure the performance of software implemented convolution filter, calculate required acceleration vs.

software implementation for given performance constraints, and Estimate the performance of hardware accelerator before implementation.

The 2D convolution is sum-of-product which can be calculated by the selecting pixel and the filter. We find the area whose size is the same as filter. Then, calculating sum-of-product to get the output value of the selecting pixel.



If we want to generate the 1080p HD Video, the performance requirements for 1080p HD video can be easily calculated. The required throughput to meet 60 FPS performance turns out to be 373 MB/s (each pixel is 8-bits (0~255)).

```
Video Resolution        = 1920 x 1080
Frame Width (pixels)    = 1920
Frame Height (pixels)   = 1080
Frame Rate(FPS)         = 60
Pixel Depth(Bits)       = 8
Color Channels(YUV)     = 3
Throughput(Pixel/s)   = Frame Width * Frame Height * Channels * FPS
Throughput(Pixel/s)   = 1920*1080*3*60
Throughput (MB/s)     = 373 MB/s
```

From the result of the first part, the required acceleration rate to generate 1080p HD video can be easily calculated. The baseline (CPU) only get 3.24 FPS. The implementation needs to be accelerated by a factor of 19x to achieve 60 FPS.

```
Number of runs    : 60
Image width       : 1920
Image height      : 1080
Filter type       : 6

Generating a random 1920x1080 input image
Running Software version on 60 images

CPU  Time         :    17.6637 s
CPU  Throughput   :    20.1519 MB/s
```

$$\text{Acceleration Factor} = \text{Throughput (Required)}/\text{Throughput(SW only)}$$

$$\text{Acceleration Factor} = 373/20.15 = 18.5\text{x}$$

We can also estimate the hardware performance. We need to consider the code structure, the filter matrix size, and the acceleration card frequency.

```
void Filter2D(
        const char          coeffs[FILTER_V_SIZE][FILTER_H_SIZE],
        float               factor,
        short               bias,
        unsigned short      width,
        unsigned short      height,
        unsigned short      stride,
        const unsigned char *src,
        unsigned char       *dst)
{
    for(int y=0; y<height; ++y)
    {
        for(int x=0; x<width; ++x)
        {
            // Apply 2D filter to the pixel window
            int sum = 0;
            for(int row=0; row<FILTER_V_SIZE; row++)
            {
                for(int col=0; col<FILTER_H_SIZE; col++)
                {
                    unsigned char pixel;
                    int xoffset = (x+col-(FILTER_H_SIZE/2));
                    int yoffset = (y+row-(FILTER_V_SIZE/2));
                    // Deal with boundary conditions : clamp pixels to 0 when outside of image
                    if ( (xoffset<0) || (xoffset>=width) || (yoffset<0) || (yoffset>=height) ) {
                        pixel = 0;
                    } else {
                        pixel = src[yoffset*stride+xoffset];
                    }
                    sum += pixel*coeffs[row][col];
                }
            }

            // Normalize and saturate result
            unsigned char outpix = MIN(MAX((int(factor * sum)+bias), 0), 255);

            // Write output
            dst[y*stride+x] = outpix;
        }
    }
}
```

The core compute is done in a 4-level nested loop, but it can break into the computation of each output pixel. The filter matrix size is 15*15. The acceleration card frequency of u50 is 300MHz. However, in Vitis HLS simulation, the frequency is ranging from 200MHz to 300MHz. We use 200MHz to estimate.

To simplify hardware implementation, we use the Vitis HLS to estimate. It will pipeline the innermost loop with II=1, performing only one multiply-accumulate (MAC) per cycle. The calculation of each clock performs a dot product of size 225.

$$\textbf{MACs per Cycle} = 1$$

$$\textbf{Hardware Fmax}(\textbf{MHz}) = 300$$

$$\textbf{Throughput} = \textbf{Fmax} * \textbf{Pixels produced per cycle}$$

$$= 300 * 1 = 300\ \textbf{MB/s}$$

$$\textbf{Output Memory Bandwidth} = \textbf{Fmax} * \textbf{Pixels produced per cycle}$$

$$= 300\ \textbf{MB/s}$$

$$\textbf{Input Memory Bandwidth}$$
$$= \textbf{Fmax} * \textbf{Input pixels read per output pixel}$$
$$= \textbf{300} * \textbf{225} = \textbf{67.5 GB/s}$$

Besides, using three compute units, one for each color channel. Therefore, we can calculate the acceleration factor to meet software performance and 60 FPS performance. The expected performance summary will be as follows:

$$\textbf{Throughput(estimated)}$$
$$= \textbf{Performance of Single Compute Unit} * \textbf{No. Compute Units}$$
$$= \textbf{300 x 3} = \textbf{900 MB/s}$$

$$\textbf{Acceleration Against Software Implementation} = \textbf{900/14.5} = \textbf{62x}$$

$$\textbf{Kernel Latency (per image on any color channel)}$$
$$= ((\textbf{1920} * \textbf{1080}))/\textbf{300} = \textbf{6.9 ms}$$

$$\textbf{Video Processing Rate} = (\textbf{1/Kernel Latency}) = \textbf{144 FPS}$$

$$\textbf{Acceleration Against 60FPS Performance} = \textbf{900/373} = \textbf{2.41x}$$

3. Design and Analysis of Hardware Kernel Module for 2-D Video Convolution Filter:

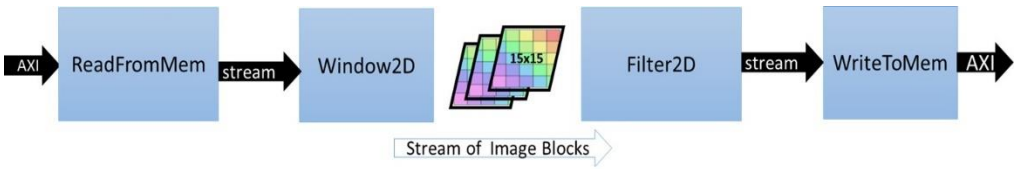
This part is about design of a convolution filter module, do performance analysis, and analyze hardware resource utilization. A bottom-up approach is followed by first developing the hardware kernel and analyzing its performance before integrating it with the host application. We will use Vitis HLS to build and estimate the performance of the kernel.

```
void Filter2DKernel(
        const char          coeffs[256],
        float               factor,
        short               bias,
        unsigned short      width,
        unsigned short      height,
        unsigned short      stride,
        const unsigned char src[MAX_IMAGE_WIDTH*MAX_IMAGE_HEIGHT],
        unsigned char       dst[MAX_IMAGE_WIDTH*MAX_IMAGE_HEIGHT])
 {
   #pragma HLS DATAFLOW

 // Stream of pixels from kernel input to filter, and from filter to output
   hls::stream<char,2>    coefs_stream;
   hls::stream<U8,2>      pixel_stream;
   hls::stream<window,3>  window_stream; // Set FIFO depth to 0 to minimize resources
   hls::stream<U8,64>     output_stream;

 // Read image data from global memory over AXI4 MM, and stream pixels out
   ReadFromMem(width, height, stride, coeffs, coefs_stream, src, pixel_stream);

   // Read incoming pixels and form valid HxV windows
   Window2D(width, height, pixel_stream, window_stream);

 // Process incoming stream of pixels, and stream pixels out
 Filter2D(width, height, factor, bias, coefs_stream, window_stream, output_stream);

 // Write an incoming stream of pixels and write them to global memory over AXI4 MM
 WriteToMem(width, height, stride, output_stream, dst);
 }
```

The top-level of the convolution filter is modeled using a dataflow process. The dataflow consists of four different functions as given above. The dataflow chain consists of four different functions as follows:

- **ReadFromMem**: reads pixel data or video input from main memory
- **Window2D**: local cache with wide(15x15 pixels) access on the output side
- **Filter2D**: core kennel filtering algorithm
- **WriteToMem**: writes output data to main memory

Two functions at the input and output read and write data from the device's global memory. The **ReadFromMem** function reads data and streams it for filtering. The **WriteToMem** function at the end of the chain writes processed pixel data to the device memory. The input data(pixels) read from the main memory is passed to the **Window2D** function, which creates a local cache and, on every cycle, provides a 15x15 pixel sample to the filter function/ block. The **Filter2D** function can consume the 15x15 pixel sample in a single cycle to perform 225(15x15) MACs per cycle.



Data Mover:

One of the advantages of using custom design hardware accelerators is the choice and architecture of custom data movers. FPGA is suited for this design. These customized data movers facilitate efficient access to global device memory and optimize bandwidth utilization by reusing data. We can build the specialized data movers, which is at the interface with main memory, at the input and output of the data processing engine or processing elements. Take convolution filter as example. It seems that to produce a single sample at the output side requires 450 memory accesses at the input side and 1 write access to the output.

Memory Accesses to Read filter Co-efficients = 15x15 = 225
Memory Accesses to Read Neighbouring Pixels = 15x15 = 225
Memory Accesses to Write to Output = 1
Total Memory Accesses = 451

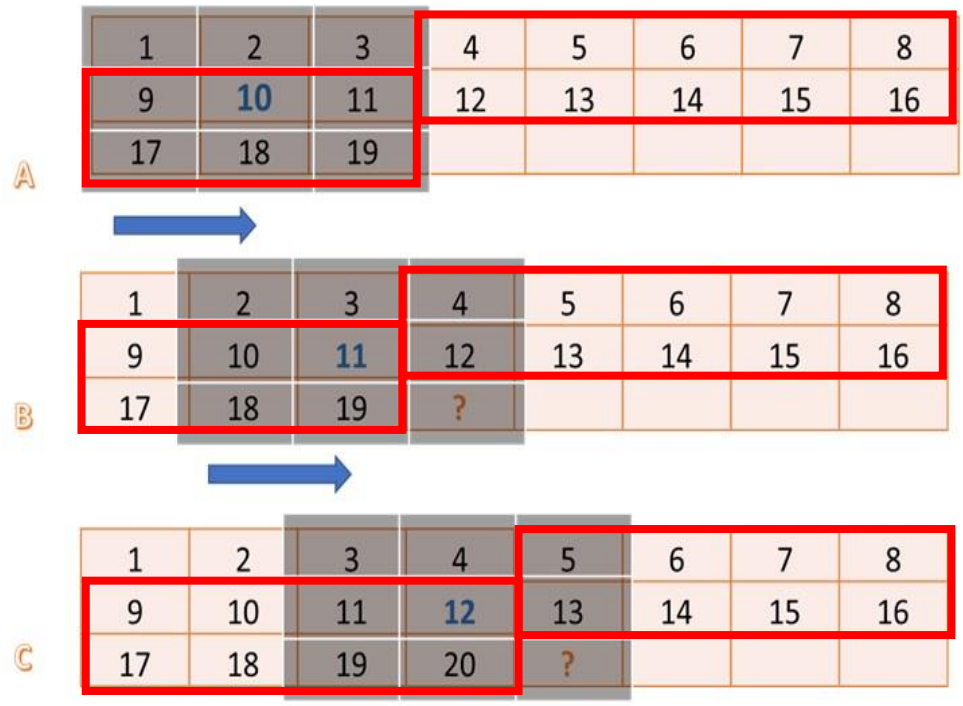
For a pure software implementation, even though many of these accesses can become fast because of caching, a large number of memory accesses will be a performance bottleneck. However, the FPGA is allowed to build the efficient

data movement and access schemes. One of the key and major advantages is the availability of substantial on-chip memory bandwidth (distributed and block memory) and the choice of a custom configuration of this bandwidth). That is, we can customize the configuration to create an on-demand cache architecture for the given algorithm.

Window2D: Line and Window Buffers:

The Window2D block is built from two blocks: **"Line buffer"** and **"Window"**.

- The line buffer is used to buffer multiple lines of a full image, and specifically, here it is designed to buffer ***FILTER_V_SIZE - 1*** image lines. Where ***FILTER_V_SIZE*** is the height of the convolution filter. The total number of pixels held by the line buffer is **(FILTER_V_SIZE-1) * MAX_IMAGE_WIDTH**.
- The "Window" block holds **FILTER_V_SIZE * FILTER_H_SIZE** pixels. Consisting of centering the filtering mask (filter coefficients) on the index of output pixel and calculating the sum-of-product (SOP).



Red block range is line buffer.

If we want to calculate the value of pixel 10. We need the 3*3 block of input pixel centered around pixel 10. In same way, we can calculate the value of pixel 11. Comparing pixel 10 and pixel 11, it has a large overlap. Only one column moves out and one column moves in. As shown in B step, as long as we want to count out the pixel 11 value, we only need to load one pixel value

owing to line buffer. Once the pixel 11 value has been counted out, the line buffer would be updated.

```
void Window2D(
        unsigned short        width,
        unsigned short        height,
        hls::stream<U8>      &pixel_stream,
        hls::stream<window>  &window_stream)
{
    // Line buffers - used to store [FILTER_V_SIZE-1] entire lines of pixels
    U8 LineBuffer[FILTER_V_SIZE-1][MAX_IMAGE_WIDTH];
#pragma HLS ARRAY_PARTITION variable=LineBuffer dim=1 complete
#pragma HLS DEPENDENCE variable=LineBuffer inter false
#pragma HLS DEPENDENCE variable=LineBuffer intra false

    // Sliding window of [FILTER_V_SIZE][FILTER_H_SIZE] pixels
    window Window;

    unsigned col_ptr = 0;
    unsigned ramp_up = width*((FILTER_V_SIZE-1)/2)+(FILTER_H_SIZE-1)/2;
    unsigned num_pixels = width*height;
    unsigned num_iterations = num_pixels + ramp_up;

    const unsigned max_iterations = MAX_IMAGE_WIDTH*MAX_IMAGE_HEIGHT + MAX_IMAGE_WIDTH*((FILTER_V_SIZE-1)/2)+(FILTER_H_SIZE-1)/2;

    // Iterate until all pixels have been processed
    update_window: for (int n=0; n<num_iterations; n++)
    {
#pragma HLS LOOP_TRIPCOUNT max=max_iterations
#pragma HLS PIPELINE II=1
```

The line buffer holds **FILTER_V_SIZE-1** lines. In general, it requires FILTER_V_SIZE lines, but a line is reduced by using the line buffer in a circular fashion and the pixels at the start of the first line buffer can be used to write new incoming pixels since they are no longer needed. The window buffer is implemented as **FILTER_V_SIZE * FILTER_H_SIZE** storage fully partitioned, giving parallel access to all elements inside the window. The data moves as a column vector of size **FILTER_V_SIZE** from line buffer to window buffer, and then this whole widow is passed through a stream to the **Filter2D** function for processing.

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP
Filter2DKernel				0.00	2211165	7.370E6	-	2211166	-	dataflow	14	139 30
ReadFromMem				0.00	2211165	7.370E6	-	2211165	-	no	0	1 2
ReadFromMem_Pipeline_read_coefs				0.00	259	863.000	-	259	-	no	0	0 1
read_coefs				-	257	857.000	3	1	256	yes	-	-
ReadFromMem_Pipeline_read_image				0.00	2210763	7.368E6	-	2210763	-	no	0	0 1
read_image				-	2210761	7.368E6	3	1	2210760	yes	-	-
Window2D				0.00	2087055	6.956E6	-	2087055	-	no	14	1 2
Window2D_Pipeline_update_window				-	2087051	6.956E6	-	2087051	-	no	0	0 2
update_window				-	2087049	6.956E6	3	1	2087047	yes	-	-
entry_proc				-	0	0.0	-	0	-	no	0	0
Filter2D				0.00	2073859	6.912E6	-	2073859	-	no	0	136 16
Filter2D_Pipeline_load_coefs_VITIS_LOOP_158_1				-	227	757.000	-	227	-	no	0	0 1
load_coefs_VITIS_LOOP_158_1				-	225	750.000	2	1	225	yes	-	-
Filter2D_Pipeline_apply_filter_VITIS_LOOP_167_2				-	2073627	6.911E6	-	2073627	-	no	0	135 14
apply_filter_VITIS_LOOP_167_2				-	2073625	6.911E6	27	1	2073600	yes	-	-
WriteToMem				0.00	2210837	7.369E6	-	2210837	-	no	0	1
WriteToMem_Pipeline_write_image				0.00	2210763	7.368E6	-	2210763	-	no	0	0
write_image				-	2210761	7.368E6	3	1	2210760	yes	-	-

Here's the performance and resource estimation. It shows the use of 139 DSP essentially for the SOP operations by the top-level module, and the use of 14 BRAMs by the Window2D data mover block. Another thing that verifies that the kernel can achieve one output sample per cycle throughput is the loop initiation intervals (II). The synthesis report expanded view shows that all loops have II=1.

4. Building the 2-D Convolution Kernel and Host Application:

This part will focus on building a hardware kernel using the Vitis application acceleration development flow. A host-side application will be implemented to coordinate all the data movements and execution triggers for controlling the kernel. During this part, real performance measurements will be taken and compared to estimated performance and the CPU-only performance.

Host application:

There are two files namely "host.cpp" and "host_randomized.cpp". They can be used to build two different versions of the host application. The way they interact with the kernel compute unit is exactly the same except that one uses the pgm image file as input. This file is repeated multiple times to emulate an image sequence(video). The randomized host uses a randomly generated image sequence. The host with random input image generation has no dependencies.

In contrast, the host code in "host.cpp" uses OpenCV libraries, specifically using OpenCV 2.4 libraries to load, unload and convert between raw image formats. In the lab, we use “host_randomized.cpp” as host code. We can modify the file in “make_options.mk” to set the parameters. Note: The default platform is u200 and trace ddr is DDR[3]. However, we use u50 to accelerate, u50 only has HBM rather than DDR. Besides, “krnl_build_options.cfg” should change into HBM as well.

```
CmdLineParser parser;
parser.addSwitch("--nruns",    "-n", "Number of times to image is processed", "1");
parser.addSwitch("--fpga",     "-x", "FPGA binary (xclbin) file to use");
parser.addSwitch("--width",    "-w", "Image width",  "1920");
parser.addSwitch("--height",   "-h", "Image height", "1080");
parser.addSwitch("--filter",   "-f", "Filter type (0-6)", "0");
parser.addSwitch("--maxreqs",  "-r", "Maximum number of outstanding requests", "3");
parser.addSwitch("--compare",  "-c", "Compare FPGA and SW performance", "false", true);
```

```
make_options.mk
1  ############## Default Vitis Build Target : Select only ONE  #
2  TARGET ?=hw
3  PLATFORM ?= xilinx_u50_gen3x16_xdma_5_202210_1
4  ENABLE_STALL_TRACE ?= yes
5  TRACE_DDR=HBM[3]
6
7  ############## Host Application Options
8  FILTER_TYPE :=3
9  PARALLEL_ENQ_REQS :=6
10 NUM_IMAGES :=1000
11 INPUT_TYPE :=random
12 INPUT_IMAGE :=../test_images/xilinx_versal_1080p.bmp
13 ENABLE_PROF?=yes
14 PROFILE_ALL_IMAGES?=no
15 IMAGE_WIDTH :=1920
16 IMAGE_HEIGHT :=1080
17 ############## Extended Options  host options
18 NUM_IMAGES_SW_EMU := 1
19 NUM_IMAGES_HW_EMU := 1
20
21 ############ pre-built xclbin path and options
22 USE_PRE_BUILT_XCLBIN := 0
23 PRE_BUILT_XCLBIN_PATH :=../xclbin/fpgabinary.hw.xclbin
24
25 ############## OpenCV Installation Paths
26 #OPENCV_INCLUDE :=/wrk/xsjhdnobkup3/shahzadb/d_apps/softwareInstall/anacondaInstall5aug20/envs/opencv2.4/include
27 #OPENCV_LIB :=/wrk/xsjhdnobkup3/shahzadb/d_apps/softwareInstall/anacondaInstall5aug20/envs/opencv2.4/lib
28 OPENCV_INCLUDE :=/usr/include/opencv2
29 OPENCV_LIB :=/usr/lib64
30
31 ############## Kernel Configuration File
32 KERNEL_CONFIG_FILE :=krnl_build_options.cfg
33
34 ############## Different log  dirs
35 VPP_TEMP_DIRS :=vpp_temp_dir
36 VPP_LOG_DIRS :=vpp_log_dir
```

After parsing the command-line options, the host application creates an OpenCL context, reads and loads the .xclbin, and creates a command queue with out-of-order execution and profiling enabled. After that, memory allocation is done, and the input image is read (or randomly generated).

After the setup is complete, the application creates a **Filter2DDispatcher** object and uses it to dispatch filtering requests on several images. The **Filter2DDispatcher** and **Filter2DRequest** manage and coordinate the execution of filtering operations on multiple compute units.

2D Filter Request:

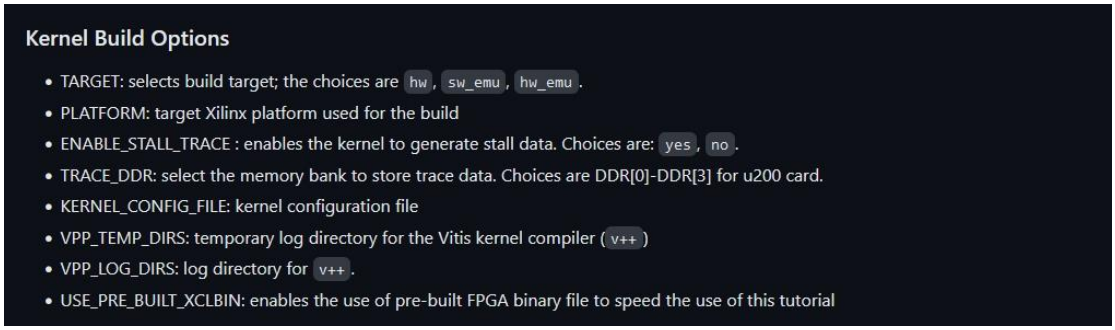
The **Filter2DRequest** class is used by the filtering request dispatcher class. An object of this class encapsulates a single request to process a single color channel (YUV) for a given image. After an object of the **Filter2DRequest** class is created, it can be used to make a call to the Filter2D method. This call will enqueue all the operations, moving input data or filter coefficients, kernel calls, and reading of output data back to the host.

2D Filter Dispatcher:

The **Filter2DDispatcher** is a container class that essentially holds a vector of request objects. The number of **Filter2DRequest** objects is defined as the max parameter for the dispatcher class at construction time. The minimum value of parameter can be as small as the number of compute units to allow at least one kernel enqueue call per compute unit to happen in parallel. However, we expect the higher value to since input and output data transfers can be overlapped between host and device.

Build the application:

The host application can be built using the **"Makefile"**. As mentioned earlier, the host application has two versions: the first version takes input images to process, the second can generate random data that will be processed as images. The top-level "**Makefile"** includes a file called "**make_options.mk".** We can change the parameter in the file to generate different host builds and kernel versions. It also provides a way to launch emulation with a specific number of test images.



Host Build Options

- ENABLE_PROF: Enables OpenCL profiling for the host application
- OPENCV_INCLUDE: OpenCV include directory path
- OPENCV_LIB: OpenCV lib directory path

Application Runtime Options

- FILTER_TYPE: selects between 6 different filter types: choices are 0-6(Identity, Blur, Motion Blur, Edges, Sharpen, Gaussian, Emboss)
- PARALLEL_ENQ_REQS: application command-line argument for parallel enqueued requests
- NUM_IMAGES: number of images to process
- IMAGE_WIDTH: image width to use
- IMAGE_HEIGHT: image height to use
- INPUT_TYPE: selects between host versions
- INPUT_IMAGE: path and name of image file
- PROFILE_ALL_IMAGES: while comparing CPU vs. FPGA, use all images or not
- NUM_IMAGES_SW_EMU: sets no. of images to use for sw_emu
- NUM_IMAGES_HW_EMU: sets no. of images to use for hw_emu

make_options.mk

```
############## Default Vitis Build Target : Select only ONE  #
TARGET ?=hw
PLATFORM ?= xilinx_u50_gen3x16_xdma_5_202210_1
ENABLE_STALL_TRACE ?= yes
TRACE_DDR=HBM[3]

############## Host Application Options
FILTER_TYPE :=3
PARALLEL_ENQ_REQS :=6
NUM_IMAGES :=1000
INPUT_TYPE :=random
INPUT_IMAGE :=../test_images/xilinx_versal_1080p.bmp
ENABLE_PROF?=yes
PROFILE_ALL_IMAGES?=no
IMAGE_WIDTH :=1920
IMAGE_HEIGHT :=1080
############## Extended Options  host options
NUM_IMAGES_SW_EMU := 1
NUM_IMAGES_HW_EMU := 1

############# pre-built xclbin path and options
USE_PRE_BUILT_XCLBIN := 0
PRE_BUILT_XCLBIN_PATH :=../xclbin/fpgabinary.hw.xclbin

############## OpenCV Installation Paths
#OPENCV_INCLUDE :=/wrk/xsjhdnobkup3/shahzadb/d_apps/softwareInstall/anacondaInstall5aug20/envs/opencv2.4/include
#OPENCV_LIB :=/wrk/xsjhdnobkup3/shahzadb/d_apps/softwareInstall/anacondaInstall5aug20/envs/opencv2.4/lib
OPENCV_INCLUDE :=/usr/include/opencv2
OPENCV_LIB :=/usr/lib64

############## Kernel Configuration File
KERNEL_CONFIG_FILE :=krnl_build_options.cfg

############## Different log  dirs
VPP_TEMP_DIRS :=vpp_temp_dir
VPP_LOG_DIRS :=vpp_log_dir
```

From our code, we are applying the random generated image whose resolution is 1920*1080 with edges filter. Then, we will run the code to perform software emulation, hardware emulation and system run. Here's the system run summary.

```
Xilinx 2D Filter Example Application (Randomized Input Version)

FPGA binary       : ./fpgabinary.hw.xclbin
Number of runs    : 60
Image width       : 1920
Image height      : 1080
Filter type       : 3
Max requests      : 6
Compare perf.     : 1

Programming FPGA device
XRT build version: 2.13.466
Build hash: f5505e402c2ca1ffe45eb6d3a9399b23a0dc8776
Build date: 2022-04-14 17:43:11
Git branch: 2022.1
PID: 13033
UID: 1059
[Sun Apr  2 08:55:56 2023 GMT]
HOST: HLS01
EXE: /mnt/HLSNAS/01.LcglIk/Vitis-Tutorials/Hardware_Acceleration/Design_Tutorials/01-convolution-tutorial/build/host.exe
[XRT] WARNING: Trace Buffer size is too big. The maximum size of 4095M will be used.
[XRT] WARNING: Trace buffer size for 0th. TS2MM is too big for memory resource.  Using 268435456 instead.
Generating a random 1920x1080 input image
Running FPGA accelerator on 60 images
Running Software version
Comparing results

Test PASSED: Output matches reference

FPGA Time         :     0.4202 s
FPGA Throughput   :   847.1229 MB/s
CPU  Time         :    17.6557 s
CPU  Throughput   :    20.1611 MB/s
FPGA Speedup      :    42.0178 x
```

We can find out the actual FPGA throughput 847MB/s is closed to our estimation 900MB/s.

Profile summary:

We can use Vitis Analyzer to analyze the system performance. The figure below shows we have 3 compute unit with clock = 300MHz (default clock). The average time is about 7 (ms), which also almost equal to our estimation.

Compute Unit Utilization

Compute Unit	Kernel	Device	Calls	Dataflow Execution	Max Parallel Executions	Dataflow Acceleration	CU Device Utilization (%)	CU Kernel Utilization (%)	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)	Clock Freq (MHz)
Filter2DKernel_1	Filter2DKernel	xilinx_u50_gen3x16_xdma_base_5-0	60	Yes	2	1.005808x	99.747	18.586	417.485	6.960	6.998	7.001	300.000
Filter2DKernel_2	Filter2DKernel	xilinx_u50_gen3x16_xdma_base_5-0	60	Yes	2	1.005818x	99.747	18.586	417.485	6.960	6.999	7.001	300.000
Filter2DKernel_3	Filter2DKernel	xilinx_u50_gen3x16_xdma_base_5-0	60	Yes	2	1.005811x	99.753	18.587	417.510	6.962	6.999	7.003	300.000

Another important measurement is the CU Utilization column, which is very close to 100 percent. This means the host was able to feed data to compute units through PCIe continuously. In other words, the host PICe bandwidth was sufficient, and compute units never saturated it.

Then, we examining the host bandwidth utilization by select **Host Data Transfers** in the report. From this table, it is clear that the host bandwidth is not fully utilized.

Host Transfer

Context:Number of Devices	Transfer Type	Number of Buffer Transfers	Transfer Rate (MB/s)	Avg Bandwidth Utilization (%)	Avg Size (KB)	Total Time (ms)	Avg Time (ms)
context0:1	READ	180	3154.644	20.025	2073.600	118.317	0.657
context0:1	WRITE	360	3765.235	23.900	1036.910	99.141	0.275

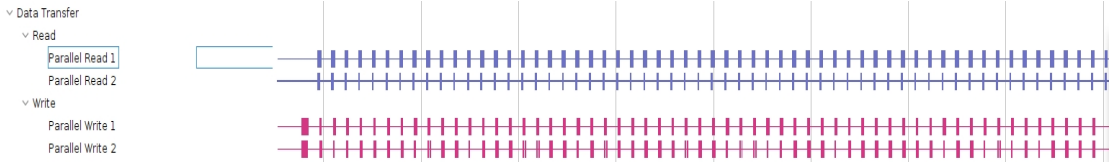
selecting **Kernel Data Transfers** in the report, you can see how much bandwidth is utilized between the kernel and the device HBM memory. You we used a single memory bank (HBM[1]) for all the compute units, as shown below.

Kernel Transfer

Compute Unit Port	Kernel Arguments	Device	Memory Resources	Transfer Type	Number of Transfers	Transfer Rate (MB/s)	Avg Bandwidth Utilization (%)	Avg Size (KB)	Avg Latency (ns)
Filter2DKernel_1/m_axi_gmem	coeffs|src|dst	xilinx_u50_gen3x16_xdma_base_5-0	HBM[1]	WRITE	121500	6273.000	32.587	1.024	48.972
Filter2DKernel_1/m_axi_gmem	coeffs|src|dst	xilinx_u50_gen3x16_xdma_base_5-0	HBM[1]	READ	121560	2043.900	10.618	1.023	152.884
Filter2DKernel_2/m_axi_gmem	coeffs|src|dst	xilinx_u50_gen3x16_xdma_base_5-0	HBM[1]	WRITE	121500	6261.930	32.530	1.024	49.058
Filter2DKernel_2/m_axi_gmem	coeffs|src|dst	xilinx_u50_gen3x16_xdma_base_5-0	HBM[1]	READ	121560	2095.600	10.886	1.023	148.981
Filter2DKernel_3/m_axi_gmem	coeffs|src|dst	xilinx_u50_gen3x16_xdma_base_5-0	HBM[1]	WRITE	121500	6322.370	32.844	1.024	48.589
Filter2DKernel_3/m_axi_gmem	coeffs|src|dst	xilinx_u50_gen3x16_xdma_base_5-0	HBM[1]	READ	121560	2069.970	10.753	1.023	152.547

The Application Timeline can also be used to examine performance parameters like CU latency per invocation and bandwidth utilization.

We can observe is the host data transfer trace as shown below. From this report, it can be seen that the host read and write bandwidth is not fully utilized as there are gaps, showing times when there are no read/write transactions occurring.



In contrast, three compute units are fully utilized relatively. There're virtually no gaps, showing almost all times are computing. In other words, all computes unit are executing when FPGA running the host code.