LabB: Convolution

Name: 吳泓毅

Student ID: R11922029

Department NTU CSIE

Github: https://github.com/Anderson-Wu/AAHLS_LAB_B

Outline

- Introduction to Convolution
- Goal of This Lab
- Performance Requirement of 1080 HD Video
- Software Implementation and Estimation
- Hardware Estimation
- Hardware Implementation
- Host Optimization
- Host Program
- Performance Analysis
- Summary

Outline

- Introduction to Convolution
  - Convolution Filter
  - Process of Filtering
- Goal of This Lab
- Performance Requirement of 1080 HD Video
- Software Implementation and Estimation
- Hardware Estimation
- Hardware Implementation
- Host Optimization
- Host Program
- Performance Analysis
- Summary

Introduction to Convolution - Convolution Filter

- A matrix of coefficients
- Different filter has different application.(e.g., filter noise, manipulate motion blur, enhance color and contrast, edge detection, etc.

Input Image

*

1	0	-1
2	0	-2
1	0	-1

Sobel x filter

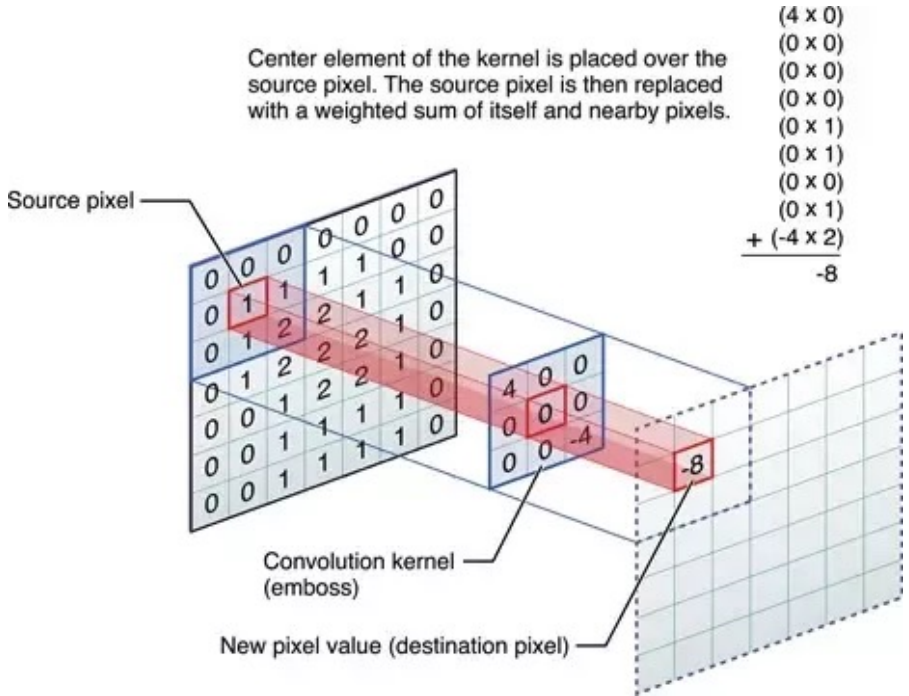
=

Vertical Edge

Introduction to Convolution - Process of Filtering

Steps

1. Selecting an input pixel as highlighted in red box in the figure below
2. Extracting a sub-matrix whose size is the same as filter coefficients
3. Calculating element-wise sum-of-product of extracted sub-matrix and coefficients matrix
4. Placing the sum-of-product as output pixel in output image/frame on the same index as the input pixel



Outline

- Introduction to Convolution
- Goal of This Lab
- Performance Requirement of 1080 HD Video
- Software Implementation and Estimation
- Hardware Estimation
- Hardware Implementation
- Host Optimization
- Host Program
- Performance Analysis
- Summary

Goal of This Lab

- Process convolution operation on **1080p HD Video** and get 60 FPS

Outline

- Introduction to Convolution
- Goal of This Lab
- Performance Requirement of 1080 HD Video
- Software Implementation and Estimation
- Hardware Estimation
- Hardware Implementation
- Host Optimization
- Host Program
- Performance Analysis
- Summary

Performance Requirement of 1080 HD Video

- 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 x Frame Height x Channels x FPS
- Throughput(Pixel/s): 1920x1080x3x60
- Throughput (MB/s): 373 MB/s

Outline

- Introduction to Convolution
- Goal of This Lab
- Performance Requirement of 1080 HD Video
- Software Implementation and Estimation
  - Convolution Operation Function
  - Top-Level Function
  - Result
- Hardware Estimation
- Hardware Implementation
- Host Optimization
- Host Program
- Performance Analysis
- Summary

Software Implementation – Convolution Operation Function

- Outer two loops: Define the pixel to be processed
- Inner two loops: Perform the sum of product operation

```
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;
    }
}
```

Software Implementation – Top-Level Function

- Top-level function calls the convolution filter function for an image with three components or channels.
- OpenMP pragma is used to parallelize software execution using multiple threads.

```
#pragma omp parallel for num_threads(3)
for(unsigned int n=0; n<numRunsSW; n++)
{
  // Compute reference results
  Filter2D(filterCoeffs[filterType], factor, bias, width, height, stride, y_src, y_ref);
  Filter2D(filterCoeffs[filterType], factor, bias, width, height, stride, u_src, u_ref);
  Filter2D(filterCoeffs[filterType], factor, bias, width, height, stride, v_src, v_ref);
}
```

Software Implementation – Result

- Acceleration Factor = $\frac{\text{Throughput (Required)}}{\text{Throughput(SW only)}} = \frac{373}{20.17} = 18.49$

```
--------------------------------------------------------------------------------
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         :    17.6464 s
CPU  Throughput   :    20.1717 MB/s
--------------------------------------------------------------------------------
```

Outline

- Introduction to Convolution
- Goal of This Lab
- Performance Requirement of 1080 HD Video
- Software Implementation and Estimation
- Hardware Estimation
  - Observation
  - Baseline Performance
  - Optimized Performance (unroll loop)
  - Optimized Performance (compute units)
- Hardware Implementation
- Host Optimization
- Host Program
- Performance Analysis
- Summary

Hardware Implementation – Observation

- The core compute is done in a 4-level nested loop, but you can break it to the compute per output pixel produced.
- In terms of the output-pixels produced, it is clear from the filter source code that a single output pixel is produced when the inner two loops finish execution once.
- These two loops are essentially doing the sum-of-product on a coefficient matrix and image submatrix. The matrix sizes are defined by the coefficient matrix, which is 15x15.
- The inner two loops are performing a dot product of size 225(15x15). In other words, the two inner loops perform 225 multiply-accumulate (MAC) operations for every output pixel produced.

Hardware Implementation – Baseline Performance

Throughput estimation(HLS will pipeline innermost loop with II=1)

- MACs per Cycle = 1
- Hardware Fmax(MHz) = 300
- Throughput = $\frac{300}{225}$ = 1.33 (MPixels/s) = 1.33 MB/s

Memory bandwidth requirements

- Output Memory Bandwidth = Throughput = 1.33 MB/s
- Input Memory Bandwidth = Throughput x 225 = 300 MB/s

Acceleration Factor = $\frac{\text{Throughput (Required)}}{\text{Throughput(SW only)}} = \frac{373}{1.33} = 280$

Hardware Implementation – Optimized Performance (unroll loop)

- We can unroll inner two loops with pipeline to gain in performance by 225(15x15), which means a throughput of 1 output pixel per cycle.

Throughput estimation

- Throughput = Fmax x Pixels produced per cycle = 300 x 1 = 300 MB/s

Memory bandwidth requirements

- Output Memory Bandwidth = Fmax x Pixels produced per cycle = 300 MB/s
- Input Memory Bandwidth = Fmax x Input pixels read per output pixel = 300 * 225 = 67.5 GB/s

The input memory bandwidth is too large. We can reduced bandwidth to around 300MB/s by using caching scheme since the convolution filter belongs to a class of kernels known as stencil kernels.

Hardware Implementation – Optimized Performance (compute units)

- Increase the number of compute units so that we can process data in parallel. We can process all color channels (YUV) on separate compute units.

Throughput estimation

- Throughput = Performance of Single Compute Unit x No. Compute Units = 300 x 3 = 900 MB/s

Acceleration Against Software Implementation = $\frac{900}{20.17} = 44.6$

Kernel Latency (per image on any color channel) = $\frac{1920x1080}{300}$ = 6.9ms

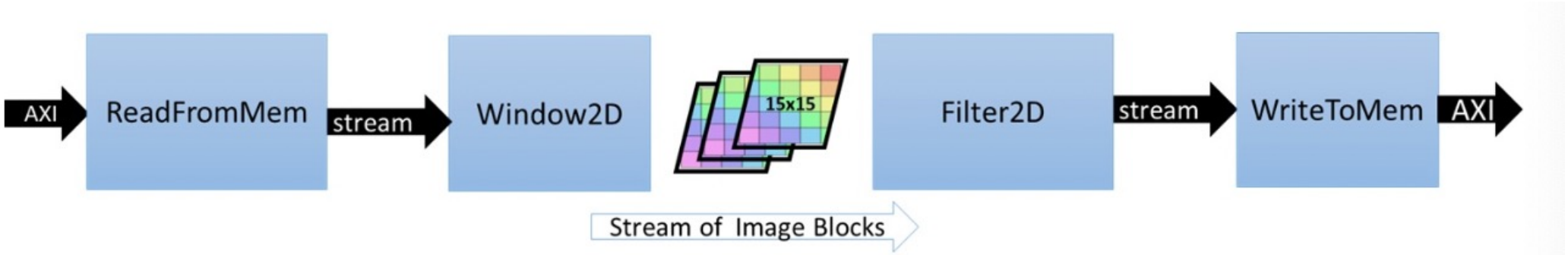
Video Processing Rate = $\frac{1}{\text{Kernel Latency}}$ = 144 FPS

Outline

- Introduction to Convolution
- Goal of This Lab
- Performance Requirement of 1080 HD Video
- Software Implementation and Estimation
- Hardware Estimation
- Hardware Implementation
  - Functions in Dataflow
  - Top-Level Function
  - Function ReadFromMem
  - Function Window2D
  - Function Filter2D
  - Function WriteToMem
- Host Optimization
- Host Program
- Performance Analysis
- Summary

Hardware Implementation – Functions in Dataflow

- 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

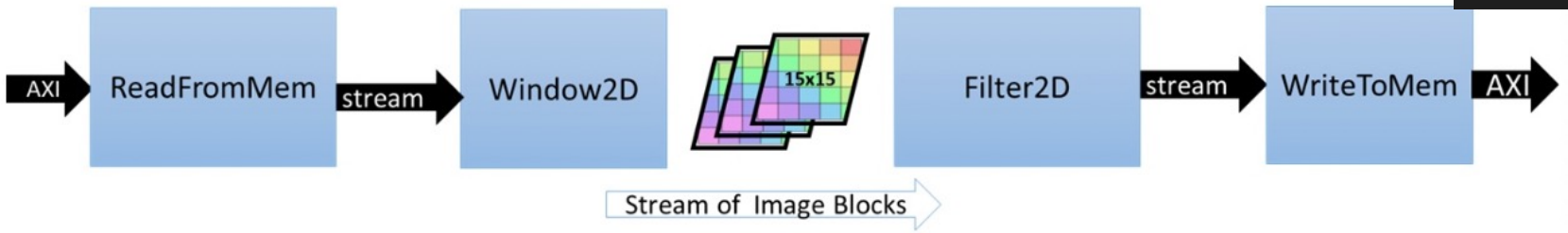


Hardware Implementation – Top-Level Function

#pragma HLS DATAFLOW

- Use a dataflow process

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

 }
```



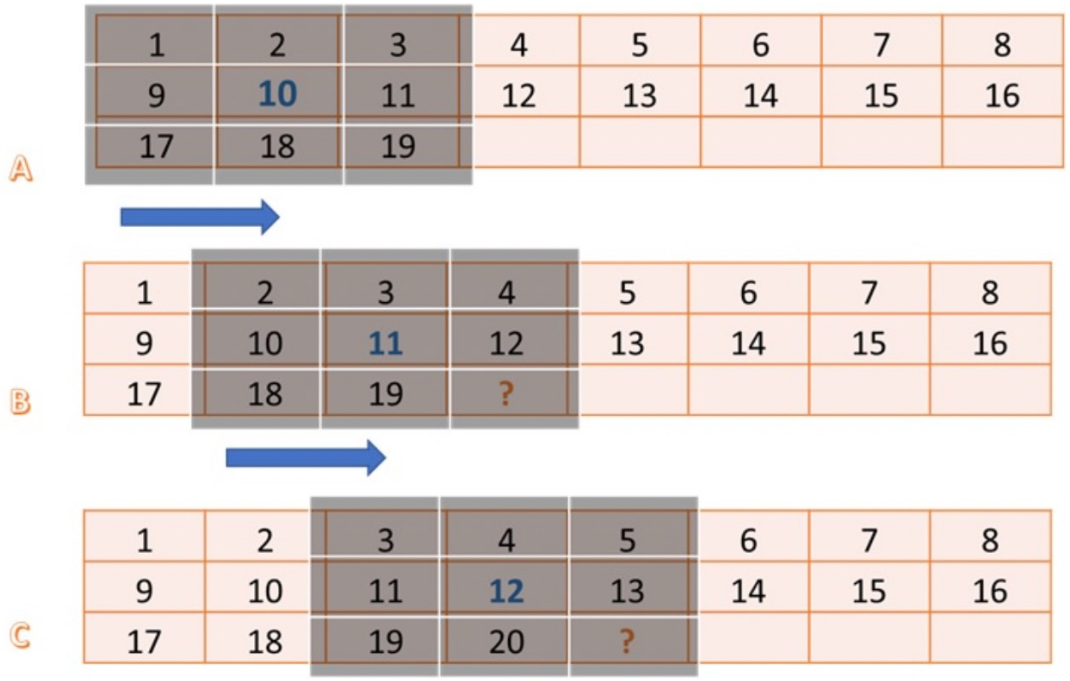
Hardware Implementation – Function ReadFromMem

- Reads pixel data or video input from main memory

```
void ReadFromMem(
        unsigned short       width,
        unsigned short       height,
        unsigned short       stride,
        const char          *coeffs,
        hls::stream<char>   &coeff_stream,
        const unsigned char *src,
        hls::stream<U8>     &pixel_stream )
{
    assert(stride <= MAX_IMAGE_WIDTH);
    assert(height <= MAX_IMAGE_HEIGHT);
    assert(stride%64 == 0);

    unsigned num_coefs = FILTER_V_SIZE*FILTER_H_SIZE;
    unsigned num_coefs_padded = (((num_coefs-1)/64)+1)*64; // Make sure number of reads of multiple of 64, enables auto-widening
    read_coefs: for (int i=0; i<num_coefs_padded; i++) {
        U8 coef = coeffs[i];
        if (i<num_coefs) coeff_stream.write( coef );
    }

    stride = (stride/64)*64; // Makes compiler see that stride is a multiple of 64, enables auto-widening
    unsigned offset = 0;
    unsigned x = 0;
    read_image: for (int n = 0; n < height*stride; n++) {
        U8 pix = src[n];
        if (x<width) pixel_stream.write( pix );
        if (x==(stride-1)) x=0; else x++;
    }
}
```

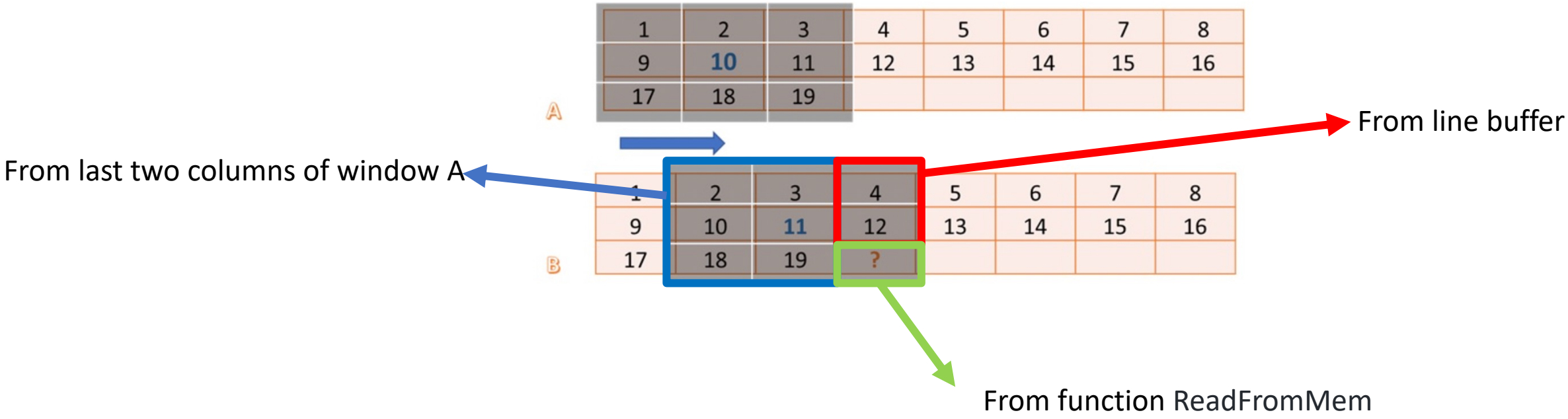
Hardware Implementation – Function Window2D

- Built from two basic blocks: the first is called a ”Window", and the second is called a ”line buffer".



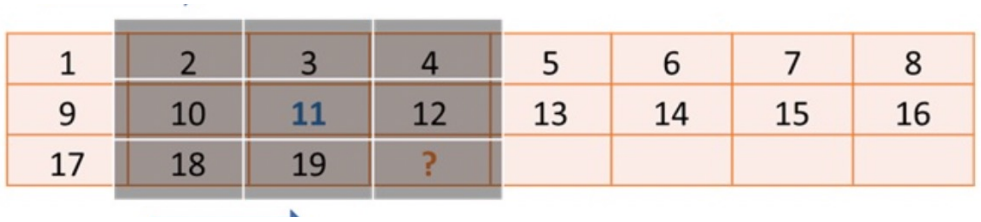
Hardware Implementation – Function Window2D - Window

- Holds FILTER_V_SIZE x FILTER_H_SIZE pixels.
- When we shift window 1 pixel, two columns will retain for the next operation, and the last column of pixels will come from line buffer and new input from function ReadFromMem. So for a new operation, we only need to get one more input by using caching method.



Hardware Implementation – Function Window2D – Line Buffer

- Used 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) x MAX_IMAGE_WIDTH.



1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	?				

Line buffer before

9	10	11	4	5	6	7	8
17	18	19	12	13	14	15	16

Line buffer after

9	10	11	12	5	6	7	8
17	18	19	20	13	14	15	16

4 is removed since it will never be used, and 20(new input) is added to line buffer

Hardware Implementation – Function Window2D – Code

```
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

        // Read a new pixel from the input stream
        U8 new_pixel = (n<num_pixels) ? pixel_stream.read() : 0;

        // Shift the window and add a column of new pixels from the line buffer
        for(int i = 0; i < FILTER_V_SIZE; i++) {
            for(int j = 0; j < FILTER_H_SIZE-1; j++) {
                Window.pix[i][j] = Window.pix[i][j+1];
            }
            Window.pix[i][FILTER_H_SIZE-1] = (i<FILTER_V_SIZE-1) ? LineBuffer[i][col_ptr] : new_pixel;
        }

        // Shift pixels in the column of pixels in the line buffer, add the newest pixel
        for(int i = 0; i < FILTER_V_SIZE-2; i++) {
            LineBuffer[i][col_ptr] = LineBuffer[i+1][col_ptr];
        }
        LineBuffer[FILTER_V_SIZE-2][col_ptr] = new_pixel;

        // Update the line buffer column pointer
        if (col_ptr==(width-1)) {
            col_ptr = 0;
        } else {
            col_ptr++;
        }

        // Write output only when enough pixels have been read the buffers and ramped-up
        if (n>=ramp_up) {
            window_stream.write(Window);
        }

    }
}
```

Hardware Implementation – Function Window2D – Code Part1

#pragma HLS ARRAY_PARTITION variable=LineBuffer dim=1 complete

- Full partition of LineBuffer on dimension 1

#pragma HLS DEPENDENCE variable=LineBuffer inter false

- Tell HLS compiler that LineBuffer has no data dependency between two loops

#pragma HLS DEPENDENCE variable=LineBuffer intra false

- Tell HLS compiler that LineBuffer has no data dependency in one loop

```
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;
```

Hardware Implementation – Function Window2D – Code Part2

#pragma HLS LOOP_TRIPCOUNT max=max_iterations

- Define max iterations of for loop, help for analysis

#pragma HLS PIPELINE II=1

- Specifies the desired initiation interval in pipeline be 1

```
    // 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

        // Read a new pixel from the input stream
        U8 new_pixel = (n<num_pixels) ? pixel_stream.read() : 0;

        // Shift the window and add a column of new pixels from the line buffer
        for(int i = 0; i < FILTER_V_SIZE; i++) {
            for(int j = 0; j < FILTER_H_SIZE-1; j++) {
                Window.pix[i][j] = Window.pix[i][j+1];
            }
            Window.pix[i][FILTER_H_SIZE-1] = (i<FILTER_V_SIZE-1) ? LineBuffer[i][col_ptr] : new_pixel;
        }

        // Shift pixels in the column of pixels in the line buffer, add the newest pixel
        for(int i = 0; i < FILTER_V_SIZE-2; i++) {
            LineBuffer[i][col_ptr] = LineBuffer[i+1][col_ptr];
        }
        LineBuffer[FILTER_V_SIZE-2][col_ptr] = new_pixel;

        // Update the line buffer column pointer
        if (col_ptr==(width-1)) {
            col_ptr = 0;
        } else {
            col_ptr++;
        }

        // Write output only when enough pixels have been read the buffers and ramped-up
        if (n>=ramp_up) {
            window_stream.write(Window);
        }

    }
}
```

Hardware Implementation – Function Filter2D

- Core kernel filtering algorithm
- Get submatrix from function Window2D and coefficients matrix from function ReadFromMem

Hardware Implementation – Function Filter2D – Code

```
void Filter2D(
        unsigned short       width,
        unsigned short       height,
        float                factor,
        short                bias,
        hls::stream<char>   &coeff_stream,
        hls::stream<window> &window_stream,
        hls::stream<U8>     &pixel_stream )
{
    assert(width  <= MAX_IMAGE_WIDTH);
    assert(height <= MAX_IMAGE_HEIGHT);

    // Filtering coefficients
    char coeffs[FILTER_V_SIZE][FILTER_H_SIZE];
#pragma HLS ARRAY_PARTITION variable=coeffs complete dim=0

    // Load the coefficients into local storage
    load_coefs: for (int i=0; i<FILTER_V_SIZE; i++) {
        for (int j=0; j<FILTER_H_SIZE; j++) {
#pragma HLS PIPELINE II=1
            coeffs[i][j] = coeff_stream.read();
        }
    }
```

```
    // Process the incoming stream of pixel windows
    apply_filter: for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
#pragma HLS PIPELINE II=1
            // Read a 2D window of pixels
            window w = window_stream.read();

            // Apply filter to the 2D 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 = w.pix[row][col];
                    }
                    sum += pixel*(char)coeffs[row][col];
                }
            }

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

            // Write the output pixel
            pixel_stream.write(outpix);
        }
    }
}
```

Hardware Implementation – Function Filter2D – Code Part1

#pragma HLS ARRAY_PARTITION variable=coeffs complete dim=0

- Full partition coefficients array to meet convolution operation's pipeline requirement

#pragma HLS PIPELINE II = 1

- Specifies the desired initiation interval in pipeline be 1

```
void Filter2D(
        unsigned short       width,
        unsigned short       height,
        float                factor,
        short                bias,
        hls::stream<char>   &coeff_stream,
        hls::stream<window> &window_stream,
        hls::stream<U8>     &pixel_stream )
{
    assert(width  <= MAX_IMAGE_WIDTH);
    assert(height <= MAX_IMAGE_HEIGHT);

    // Filtering coefficients
    char coeffs[FILTER_V_SIZE][FILTER_H_SIZE];
#pragma HLS ARRAY_PARTITION variable=coeffs complete dim=0

    // Load the coefficients into local storage
    load_coefs: for (int i=0; i<FILTER_V_SIZE; i++) {
        for (int j=0; j<FILTER_H_SIZE; j++) {
#pragma HLS PIPELINE II=1
            coeffs[i][j] = coeff_stream.read();
        }
    }
```

Hardware Implementation – Function Filter2D – Code Part2

#pragma HLS PIPELINE II=1

- Specifies the desired initiation interval in pipeline be 1

For loops under #pragma HLS PIPELINE are all be unrolled, and with II = 1, we can produce 1 pixel output per cycle

```
    // Process the incoming stream of pixel windows
    apply_filter: for (int y = 0; y < height; y++)
    {
        for (int x = 0; x < width; x++)
        {
#pragma HLS PIPELINE II=1
            // Read a 2D window of pixels
            window w = window_stream.read();

            // Apply filter to the 2D 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 = w.pix[row][col];
                    }
                    sum += pixel*(char)coeffs[row][col];
                }
            }

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

            // Write the output pixel
            pixel_stream.write(outpix);
        }
    }
}
```

Hardware Implementation – Function WriteToMem

- WriteToMem: writes output data to main memory

```
void WriteToMem(
        unsigned short       width,
        unsigned short       height,
        unsigned short       stride,
        hls::stream<U8>     &pixel_stream,
        unsigned char       *dst)
{
    assert(stride <= MAX_IMAGE_WIDTH);
    assert(height <= MAX_IMAGE_HEIGHT);
    assert(stride%64 == 0);

    stride = (stride/64)*64; // Makes compiler see that stride is a multiple of 64, enables auto-widening
    unsigned offset = 0;
    unsigned x = 0;
    write_image: for (int n = 0; n < height*stride; n++) {
        U8 pix = (x<width) ? pixel_stream.read() : 0;
        dst[n] = pix;
        if (x==(stride-1)) x=0; else x++;
    }
}
```

Outline

- Introduction to Convolution
- Goal of This Lab
- Performance Requirement of 1080 HD Video
- Software Implementation and Estimation
- Hardware Estimation
- Hardware Implementation
- Host Optimization
- Host Program
- Performance Analysis
- Summary

Host Optimization

- In file krnl_build_options.cfg, we set the compute unit be three, so we can run three kernels at the same time.
- Use outstanding requests to increase bus utilization and compute units utilization which means compute units always have data to execute.

```
[connectivity]
nk=Filter2DKernel:3
sp=Filter2DKernel_1.m_axi_gmem:HBM[0]
sp=Filter2DKernel_2.m_axi_gmem:HBM[0]
sp=Filter2DKernel_3.m_axi_gmem:HBM[0]
~
~
```

Outline

- Introduction to Convolution
- Goal of This Lab
- Performance Requirement of 1080 HD Video
- Software Implementation and Estimation
- Hardware Estimation
- Hardware Implementation
- Host Optimization
- Host Program
  - Command Line Arguments
  - Main Function
  - Filter2DDispatcher Class
  - Filter2DRequest Class
- Performance Analysis
- Summary

Host Program – Command Line Arguments

CmdLineParser parser;

- parser.addSwitch("--nruns", "-n", "Number of times the image is processed", "1");
- parser.addSwitch("--fpga", "-x", "FPGA binary (xclbin) file to use");
- parser.addSwitch("--input", "-i", "Input image file");
- 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);

- parser.addSwitch("--width", "-w", "Image width", "1920");
- parser.addSwitch("--height", "-h", "Image height", "1080");

Host Program – Main Function

Get Xilinx device, create context, program, and command queue

```
printf("Programming FPGA device\n");
cl_int err;
std::vector<cl::Device> devices = xcl::get_xil_devices();
devices.resize(1); // (arbitrarily) use the first Xilinx device that is found
OCL_CHECK(err, cl::Context context(devices[0], NULL, NULL, NULL, &err));
unsigned fileBufSize;
char* fileBuf = xcl::read_binary_file(fpgaBinary.c_str(), fileBufSize);
cl::Program::Binaries bins{{fileBuf, fileBufSize}};
OCL_CHECK(err, cl::Program program(context, devices, bins, NULL, &err));
OCL_CHECK(err, cl::CommandQueue queue(context, devices[0], cl::QueueProperties::Profiling | cl::QueueProperties::OutOfOrder, &err));
```

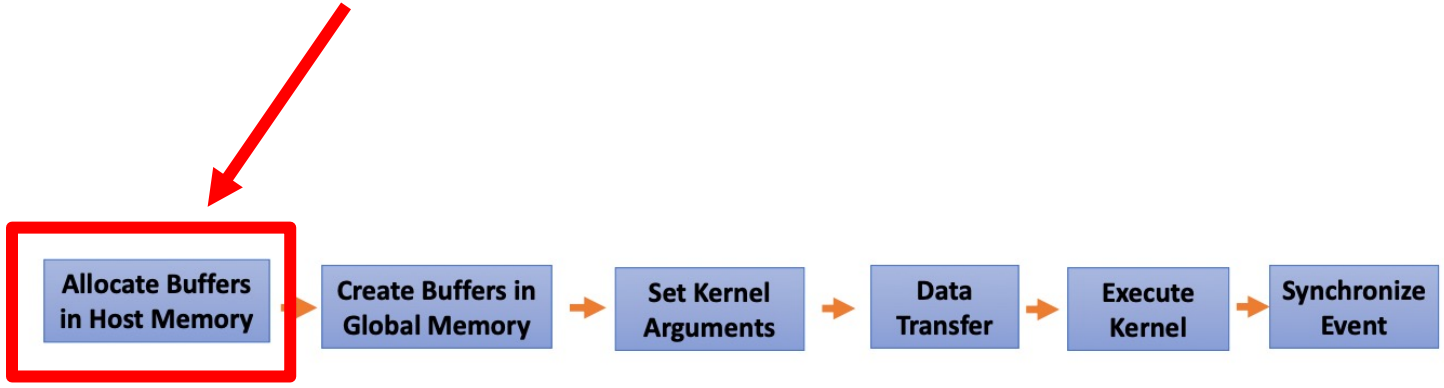
Host Program – Main Function

Create host memory and generate images

```
// Read Input image
unsigned int stride    = ceil(width/64.0)*64;
unsigned int nbytes    = (stride*height);

// Input and output buffers (Y,U, V)
unsigned char *y_src = (unsigned char *)malloc(nbytes);
unsigned char *u_src = (unsigned char *)malloc(nbytes);
unsigned char *v_src = (unsigned char *)malloc(nbytes);
unsigned char *y_dst = (unsigned char *)malloc(nbytes);
unsigned char *u_dst = (unsigned char *)malloc(nbytes);
unsigned char *v_dst = (unsigned char *)malloc(nbytes);

// Generate random image data
printf("Generating a random %dx%d input image\n", width, height);
for (unsigned int i=0; i<nbytes; i++) {
      y_src[i] = rand();
      u_src[i] = rand();
      v_src[i] = rand();
      y_dst[i] = 0;
      u_dst[i] = 0;
      v_dst[i] = 0;
}

// Retrieve filter factor and bias
float factor = filterFactors[filterType];
short bias   = filterBiases[filterType];
```

Create host memory

At this stage

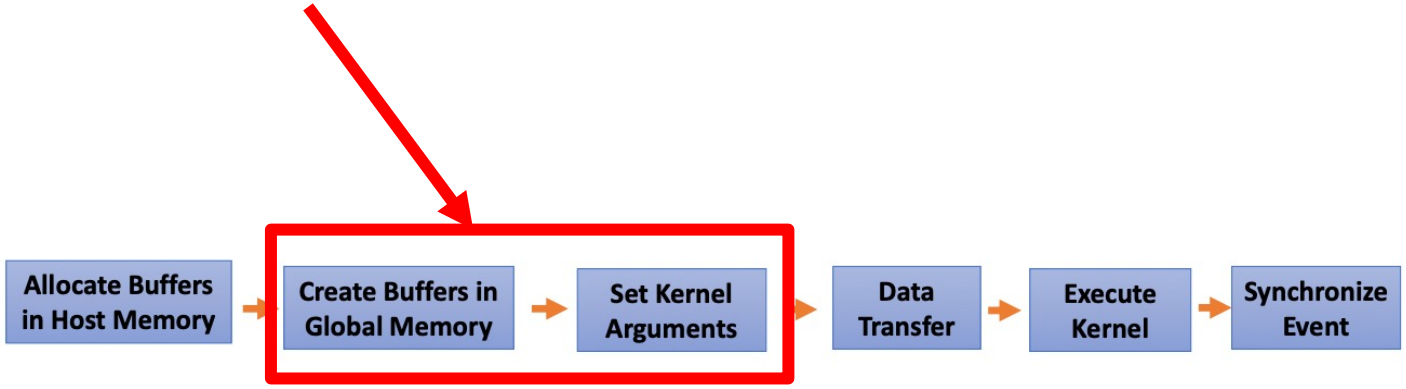


Host Program – Main Function

Create global memory and send part of kernel arguments

```
// Dispatcher of requests to the kernel
// 'maxReqs' controls the maximum number of outstanding requests to the kernel
// and equates to the depth of SW pipelining.
Filter2DDispatcher Filter2DKernel(context, program, queue, maxReqs);
```

At this stage



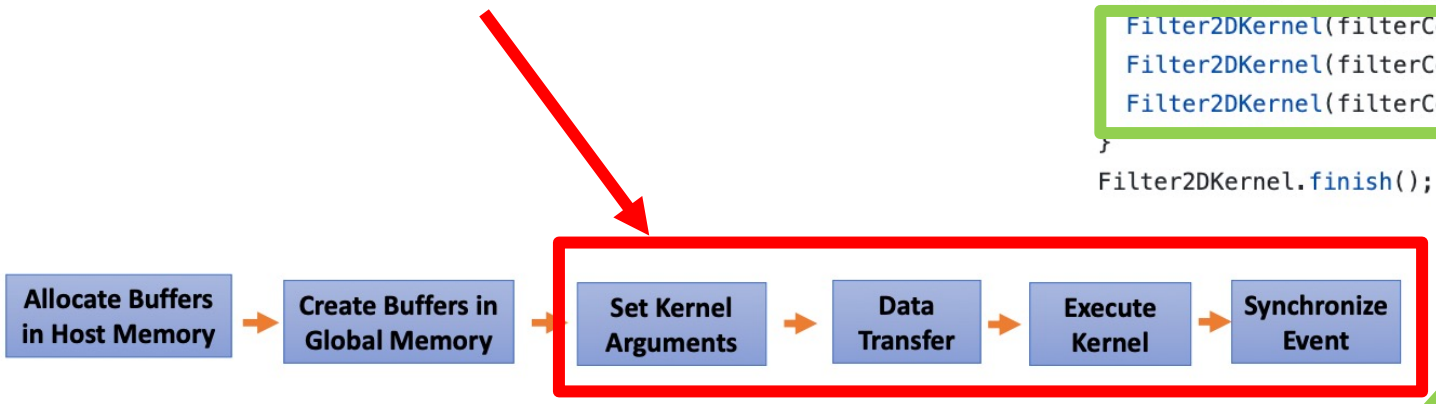
Create Filter2DDispatcher object to handle request

Host Program – Main Function

Set kernel arguments, transfer images, coefficients matrix from host memory to global memory, start kernel execution, and then get the result back to host memory

```
auto fpga_begin = std::chrono::high_resolution_clock::now();

for(unsigned int n=0; n<numRuns; n++)
{
  // Enqueue independent requests to Blur Y, U and V planes
  // Requests will run sequentially if there is a single kernel
  // Requests will run in parallel is there are two or more kernels
  Filter2DKernel(filterCoeffs[filterType], factor, bias, width, height, stride, y_src, y_dst)
  Filter2DKernel(filterCoeffs[filterType], factor, bias, width, height, stride, u_src, u_dst)
  Filter2DKernel(filterCoeffs[filterType], factor, bias, width, height, stride, v_src, v_dst)
}
Filter2DKernel.finish();
```

At this stage



Can run parallel since we have three compute units

Call Filter2D method in Filter2DRequest object to do kernel execution

Host Program – Filter2DDispatcher Class

- The top-level class that provides an end-user API to schedule and manage kernel calls.
- Holds a vector of Filter2DRequest objects. The number of request objects that are instantiated is defined as the **max** at construction time.
- This parameter's minimum value 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. But a larger value is desired since it will allow overlap between input and output data transfers happening between host and device.

Host Program – Filter2DDispatcher - Code

Store all the requests

Constructor, create Filter2DRequest Objects

Call Filter2D method in Filter2DRequest object to do kernel execution

```
class Filter2DDispatcher
{
  std::vector<Filter2DRequest> req;
  int max;
  int cnt;

  public:
    Filter2DDispatcher(cl::Context &context, cl::Program &program, cl::CommandQueue &queue, int nreqs
    {
      cnt = 0;
      max = nreqs;
      for(int i=0; i<max; i++) {
        req.push_back( Filter2DRequest(context, program, queue) );
      }
    };

    int operator () (
      const char        coeffs[FILTER_V_SIZE][FILTER_H_SIZE],
      float             factor,
      short             bias,
      unsigned short    width,
      unsigned short    height,
      unsigned short    stride,
      unsigned char    *src,
      unsigned char    *dst )
    {
      cnt++;
      req[cnt%max].Filter2D(coeffs, factor, bias, width, height, stride, src, dst)
      return (cnt%max);
    }

    void finish(int id) {
      if (id<max) {
        req[id].finish();
      }
    }

    void finish() {
      for(int i=0; i<max; i++) {
        req[i].finish();
      }
    }
};
```

Host Program – Filter2DRequest Class

- The Filter2DRequest class is used by Filter2DDispatcher class.
- 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.

Host Program – Filter2DRequest – Code Part1

Constructor of Filter2DRequest

Create kernel

Create global memory

Set kernel arguments

Can be removed by experiment

```
class Filter2DRequest
{
    cl::Kernel            kernel;
    cl::CommandQueue      q;
    cl::Buffer            coef_buffer;
    cl::Buffer            src_buffer;
    cl::Buffer            dst_buffer;
    std::vector<cl::Event> events;

  public:
    Filter2DRequest(cl::Context &context, cl::Program &program, cl::CommandQueue &queue)
    {
      cl_int err;

      q = queue;

      OCL_CHECK(err, kernel = cl::Kernel(program,"Filter2DKernel", &err))

      // Allocate input and output buffers
      OCL_CHECK(err, coef_buffer = cl::Buffer(context, CL_MEM_READ_ONLY,  (FILTER_V_SIZE*FILTER_V_SIZE)*sizeof(char), nullptr, &err));
      OCL_CHECK(err, src_buffer  = cl::Buffer(context, CL_MEM_READ_ONLY,  (1920*1080)*sizeof(char), nullptr, &err));
      OCL_CHECK(err, dst_buffer  = cl::Buffer(context, CL_MEM_WRITE_ONLY, (1920*1080)*sizeof(char), nullptr, &err));

      // Set kernel arguments - this pins the buffers to specific global memory banks
      OCL_CHECK(err, err = kernel.setArg(0, coef_buffer));
      OCL_CHECK(err, err = kernel.setArg(6, src_buffer));
      OCL_CHECK(err, err = kernel.setArg(7, dst_buffer));

      // Make buffers resident in the device
      // If done after setArg, then buffers are pinned and runtime knows in which bank they should be made resident,
      // removing the need for using the vendor extensions to explicitely map to DDR.
      OCL_CHECK(err, err = q.enqueueMigrateMemObjects({coef_buffer, src_buffer, dst_buffer}, CL_MIGRATE_MEM_OBJECT_CONTENT_UNDEFINED))

      // Make sure buffers are migrated before continuing
      q.finish();
    };
```

Host Program – Filter2DRequest – Code Part2

Filter2D method

Wait for previous request terminate

Set other kernel arguments

Transfer image and coefficients matrix from host to global memory

Enqueue task

Move result from global memory to host memory

```
void Filter2D(
  const char        coeffs[FILTER_V_SIZE][FILTER_H_SIZE],
  float             factor,
  short             bias,
  unsigned short    width,
  unsigned short    height,
  unsigned short    stride,
  unsigned char    *src,
  unsigned char    *dst )
{

  assert(width  <= 1920);
  assert(height <= 1080);
  assert(stride%64 == 0);

  cl_int    err;
  cl::Event in1_event;
  cl::Event in2_event;
  cl::Event run_event;
  cl::Event out_event;

  size_t offset = 0;
  int nbytes = stride*height*sizeof(char);

  // If a previous transaction is pending, wait until it completes
  finish();

  // Set kernel arguments - since buffers are reused, no need to set these args each time...
  // OCL_CHECK(err, err = kernel.setArg(0, coef_buffer));
  OCL_CHECK(err, err = kernel.setArg(1, factor));
  OCL_CHECK(err, err = kernel.setArg(2, bias));
  OCL_CHECK(err, err = kernel.setArg(3, width));
  OCL_CHECK(err, err = kernel.setArg(4, height));
  OCL_CHECK(err, err = kernel.setArg(5, stride));
  // OCL_CHECK(err, err = kernel.setArg(6, src_buffer));
  // OCL_CHECK(err, err = kernel.setArg(7, dst_buffer));

  // Schedule the writing of the inputs from host to device
  OCL_CHECK(err, err = q.enqueueWriteBuffer(coef_buffer, CL_FALSE, offset, (FILTER_V_SIZE*FILTER_V_SIZE)*sizeof(char), &coeffs[0][0], nullptr, &in1_event) );
  OCL_CHECK(err, err = q.enqueueWriteBuffer(src_buffer,  CL_FALSE, offset, nbytes, src, nullptr, &in2_event) );
  events.push_back(in1_event);
  events.push_back(in2_event);

  // Schedule the execution of the kernel
  OCL_CHECK(err, err = q.enqueueTask(kernel, &events, &run_event));
  events.push_back(run_event);

  // Schedule the reading of the outputs from device back to host
  OCL_CHECK(err, err = q.enqueueReadBuffer(dst_buffer, CL_FALSE, offset, nbytes, dst, &events, &out_event) );
  events.push_back(out_event);
}
```

Host Program – Filter2DRequest – Code Part3

Finish method

Waiting for end of execution

```
void finish()
{
  if (events.size()>0) {
    events.back().wait();
    events.clear();
    if (getenv("XCL_EMULATION_MODE") != NULL) {
      printf("  finished Filter2DRequest\n");
    }
  }
}
};
```

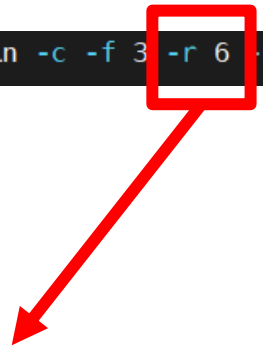
Outline

- Introduction to Convolution
- Goal of This Lab
- Performance Requirement of 1080 HD Video
- Software Implementation and Estimation
- Hardware Estimation
- Hardware Implementation
- Host Optimization
- Host Program
- Performance Analysis
  - Command to Run
  - Hardware Accelerated Result
  - Profile Summary
  - Application Timeline
- Summary

Performance Analysis – Command to Run

- Set outstanding to 6 to increase compute units utilization.

```
cd ./build && unset XCL_EMULATION_MODE && ./host.exe  -x ./fpgabinary.hw.xclbin -c -f 3 -r 6 -n 60  -w 1920 -h 1080
```



number of outstanding is 6

Performance Analysis - Hardware Accelerated Result

```
cd ./build && unset XCL_EMULATION_MODE && ./host.exe  -x ./fpgabinary.hw.xclbin -c -f 3 -r 6 -n 60  -w 1920 -h 1080
--------------------------------------------------------------------------------

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: 1983108
UID: 1060
[Fri Oct 21 04:57:10 2022 GMT]
HOST: HLS03
EXE: /mnt/HLSNAS/03.xoLHJE/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 Throughput   :   845.5195 MB/s
CPU  Throughput   :    20.0386 MB/s
FPGA  peedup      :    42.1946 x
--------------------------------------------------------------------------------
```

Meet the requirement 373MB/s

Performance Analysis – Profile Summary

- From this table, it can be seen that the kernel compute time as displayed in the **Avg Time** column is about 7 ms, almost equal to the estimated kernel latency.
- We can see that **CU Device 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.

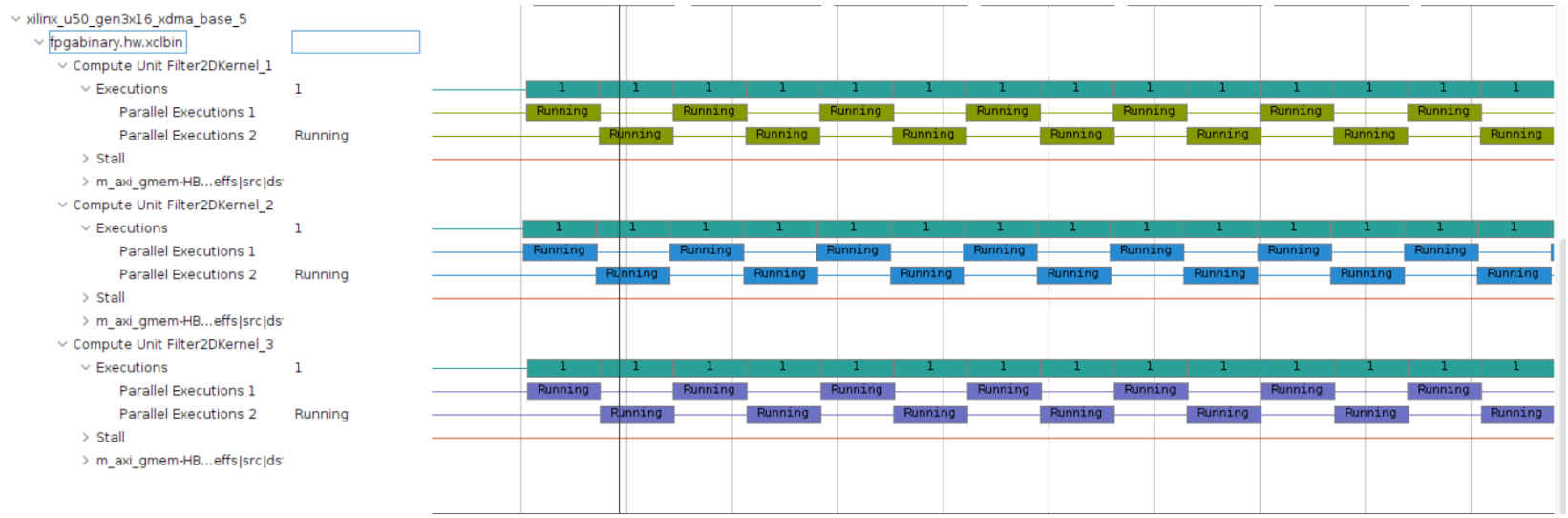
Compute Unit	Kernel	Device	Calls	Dataflow Execution	Max Parallel Executions	Dataflow Acceleration	CU Device Utilization (%)	CU Kernel Utilization (%)	Total Time (ms)	Min Time (ms)	Avg Time (ms)	Max Time (ms)	Clock Freq (MHz)
Filter2DKernel_1	Filter2DKernel	xilinx_u50_gen3x16_xdma_base_5-0	60	Yes	2	1.005834x	99.902	18.305	417.503	6.959	6.999	7.002	300.000
Filter2DKernel_2	Filter2DKernel	xilinx_u50_gen3x16_xdma_base_5-0	60	Yes	2	1.005828x	99.898	18.304	417.484	6.962	6.999	7.001	300.000
Filter2DKernel_3	Filter2DKernel	xilinx_u50_gen3x16_xdma_base_5-0	60	Yes	2	1.005789x	99.899	18.304	417.489	6.965	6.998	7.002	300.000

```
[connectivity]
nk=Filter2DKernel:3
sp=Filter2DKernel_1.m_axi_gmem:HBM[0]
sp=Filter2DKernel_2.m_axi_gmem:HBM[0]
sp=Filter2DKernel_3.m_axi_gmem:HBM[0]
~
~
```

We create three compute units in krnl_build_options.cfg

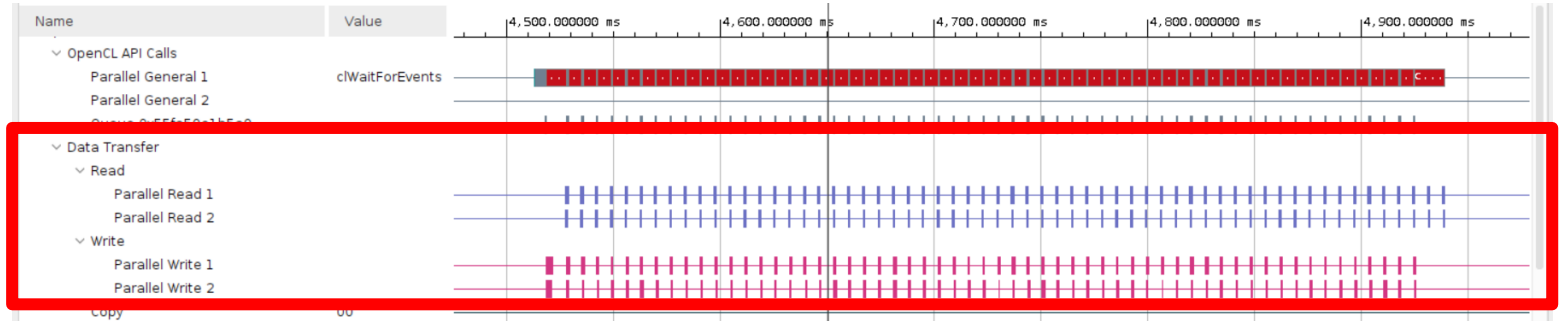
Performance Analysis – Application Timeline

- We can also see that compute units can run almost all time due to outstanding request by seeing application timeline



Performance Analysis – Application Timeline

- It can be seen that the host read and write bandwidth is not fully utilized as there are gaps which means that only a fraction of host PCIe bandwidth is utilized.



Outline

- Introduction to Convolution
- Goal of This Lab
- Performance Requirement of 1080 HD Video
- Software Implementation and Estimation
- Hardware Estimation
- Hardware Implementation
- Host Optimization
- Host Program
- Performance Analysis
- Summary

Summary

- In this lab, We have learned:
  - How to build, run and analyze the performance of a video filter
  - How to write an optimized host-side application for multi CU designs
  - How to estimate kernel performance and compare it with measured performance