1. There are 648*488 pixels in each frame and if there are 10 bits per pixel, there will be 648*488*10 = 3162240 bits = 395280 Bytes transferred per frame.
2. The total data transfer is 395280 Bytes for each frame. Because total data transfer must be integer multiple of the block size, 395280 = 2^4 * 3^4 * 5 * 61, so I would choose 1296 (2^4*3^4) as the block size so we can transfer 5 * 61 to complete the transfer for one frame.

Verilog code:
Main.v:

```
`timescale 1ns / 1ps


module Main(
    output [7:0] led,
    input sys_clkn,
    input sys_clkp,



    input  CVM300_CLK_OUT,
    output CVM300_CLK_IN,
    output CVM300_SYS_RES_N,
    output CVM300_FRAME_REQ,
    output CVM300_SPI_EN,
    output CVM300_SPI_CLK,
    input  CVM300_SPI_OUT,
    output CVM300_SPI_IN,
    input CVM300_Line_valid,
    input CVM300_Data_valid,
    input [9:0] CVM300_D,



    input  [4:0] okUH,
    output [2:0] okHU,
    inout  [31:0] okUHU,
    inout  okAA
);
```

```
    wire clk;
```

```
    IBUFGDS osc_clk(
        .O(clk),
        .I(sys_clkp),
        .IB(sys_clkn)
    );
```

```
wire [31:0]PC_rx;
wire [31:0]PC_tx;
wire [31:0]PC_command;
wire [31:0]PC_addr;
wire [31:0]PC_val;

wire FIFO_wr_clk;
wire FIFO_wr_enable;
wire [31:0]FIFO_data_in;
wire FIFO_full;
wire FIFO_BT;
wire FIFO_read_enable;
wire FIFO_read_reset;
wire FIFO_write_reset;
wire USB_ready;

//PC communication/////////////////////////////////////////////////////////////////
USB_Driver USB_Driver(
    .clk(clk),

    .okUH(okUH),
    .okHU(okHU),
    .okUHU(okUHU),
    .okAA(okAA),

    .PC_rx(PC_rx),
    .PC_tx(PC_tx),
    .PC_command(PC_command),
    .PC_addr(PC_addr),
    .PC_val(PC_val),

    .FIFO_wr_clk(FIFO_wr_clk),
    .FIFO_read_reset(FIFO_read_reset),
    .FIFO_write_reset(FIFO_write_reset),
    .FIFO_wr_enable(FIFO_wr_enable),
    .FIFO_data_in(FIFO_data_in),
    .FIFO_full(FIFO_full),
    .FIFO_BT(FIFO_BT),
    .FIFO_read_enable(FIFO_read_enable),

    .USB_ready(USB_ready));
```

```
    // PC communication//////////////////////////////////////////////////////////////
    CVM300_driver CVM300_driver (
        .clk(clk),
        .CVM300_CLK_OUT(CVM300_CLK_OUT),
        .CVM300_CLK_IN(CVM300_CLK_IN),
        .CVM300_SYS_RES_N(CVM300_SYS_RES_N),
        .CVM300_FRAME_REQ(CVM300_FRAME_REQ),
        .CVM300_SPI_EN(CVM300_SPI_EN),
        .CVM300_SPI_CLK(CVM300_SPI_CLK),
        .CVM300_SPI_OUT(CVM300_SPI_OUT),
        .CVM300_SPI_IN(CVM300_SPI_IN),
        .CVM300_LVAL(CVM300_Line_valid),
        .CVM300_DVAL(CVM300_Data_valid),
        .CVM300_D(CVM300_D),

        .PC_rx(PC_rx),
        .PC_tx(PC_tx),
        .PC_command(PC_command),
        .PC_addr(PC_addr),
        .PC_val(PC_val),

        .FIFO_wr_clk(FIFO_wr_clk),
        .FIFO_read_reset(FIFO_read_reset),
        .FIFO_write_reset(FIFO_write_reset),
        .FIFO_wr_enable(FIFO_wr_enable),
        .FIFO_data_in(FIFO_data_in),
        .FIFO_full(FIFO_full),
        .FIFO_BT(FIFO_BT),

        .USB_ready(USB_ready));
    //Instantiate the ILA module
    ila_0 ila_sample12 (
        .clk(clk),
        .probe0({CVM300_D,CVM300_Line_valid,CVM300_Data_valid,CVM300_CLK_OUT}),
        .probe1(CVM300_FRAME_REQ),
        .probe2(FIFO_BT),
        .probe3(FIFO_read_enable),
        .probe4(FIFO_wr_enable),
        .probe5(USB_ready));
endmodule
```

SPI.v:

```
`timescale 1ns / 1ps
```

```
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2024/10/19 12:59:21
// Design Name:
// Module Name: SPI
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module SPI_driver(
    input  wire clk,
    output reg [2:0] cur_state,

    input wire  SPI_MISO,
    output reg  SPI_MOSI,
    output reg  SPI_CLK,
    output reg  SPI_EN,

    output reg busy,
    input wire command_read,
    input wire rx_read,
    input wire tx_read,
    input wire [1:0] Spi_rw,
    output reg [7:0] Spi_rx_reg,
    input wire [7:0] Spi_tx_reg
    );
```

```
localparam IDLE  = 3'b000;
localparam SPITX = 3'b001;
localparam SPIRX = 3'b010;
localparam SPIED = 3'b100;
```

```
reg [1:0] command_FIFO[15:0];
reg [7:0] tx_FIFO[15:0];
reg [7:0] rx_FIFO[15:0];
reg [3:0] command_addrw;
reg [3:0] command_addrr;
reg [3:0] tx_addrw;
reg [3:0] tx_addrr;
reg [3:0] rx_addrw;
reg [3:0] rx_addrr;
wire command_empty;
wire tx_empty;
wire rx_empty;
```

```
initial begin
    cur_state = 7'd0;
    SPI_MOSI = 1'b0;
    SPI_CLK = 1'b0;
    SPI_EN = 1'b0;
    busy = 1'b0;
    command_addrw = 4'b0;
    command_addrr = 4'd0;
    tx_addrw = 4'd0;
    tx_addrr = 4'd0;
    rx_addrw = 4'd0;
    rx_addrr = 4'd0;
end
```

```
assign tx_empty = &(~(tx_addrr^tx_addrw));
assign rx_empty = &(~(rx_addrr^rx_addrw));
assign command_empty = &(~(command_addrr^command_addrw));
```

```
always @(posedge clk) begin
    if (command_read == 1'b1) begin
        command_FIFO[command_addrw] <= Spi_rw;
        command_addrw <= command_addrw + 1;
    end
    if (tx_read == 1'b1) begin
        tx_FIFO[tx_addrw] <= Spi_tx_reg;
        tx_addrw <= tx_addrw + 1;
    end
    if (rx_read == 1'b1) begin
        if(rx_empty != 1'b1) begin
            Spi_rx_reg <= rx_FIFO[rx_addrr];
            rx_addrr <= rx_addrr + 1;
```

```
        end else begin
            Spi_rx_reg <= 8'b11111111;
        end
    end
end
```

```
reg[2:0] bit_counter;
reg[2:0] clk_counter;
reg[7:0] rx_temp_reg;
```

```
always @(posedge clk) begin
        case(cur_state)
            IDLE : begin
                busy <= 1'b0;
                if(command_empty == 1'b0)begin
                    if(command_FIFO[command_addrr] == 2'b01)begin
                        command_addrr  <= command_addrr + 1;
                        cur_state      <= SPITX;
                        clk_counter    <= 3'b000;
                        bit_counter    <= 3'b111;
                        busy <= 1'b1;
                    end
                    //add error detection if the first command out of IDLE is rx, this is incorrectly set by the
controller
                    // also add error detection if when entering TX, check for TX_FIFO empty, if not, controller
is incorrectly set
                end
            end
            SPITX: begin
                case(clk_counter)
                    3'b000 : begin
                        SPI_EN   <= 1'b1;
                        SPI_CLK  <= 1'b0;
                        SPI_MOSI <= tx_FIFO[tx_addrr][bit_counter];
                        busy <= 1'b1;
                        clk_counter <= clk_counter + 1;
                    end
                    3'b100 : begin
                        SPI_CLK <= 1'b1;
                        clk_counter <= clk_counter + 1;
                    end
                    3'b111 : begin
                        if(bit_counter != 3'b000) begin
                            bit_counter <= bit_counter -1;
```

```
                    clk_counter <= 3'b000;
                end else begin
                    bit_counter <= 3'b111;
                    clk_counter <= 3'b000;
                    tx_addrr <= tx_addrr + 1;
                    if(command_empty == 1'b1)cur_state   <= SPIED;
                    else begin
                        if (command_FIFO[command_addrr] == 2'b01) cur_state <= SPITX;
                        else cur_state <= SPIRX;
                        command_addrr <= command_addrr + 1;
                        //add error detection if 2'b10 or 2'b01 is not the data read from the FIFO
                    end
                end
            end
            default : begin
                clk_counter <= clk_counter + 1;
            end
        endcase
    end
    SPIRX: begin
        case(clk_counter)
            3'b000 : begin
                SPI_EN  <= 1'b1;
                SPI_CLK <= 1'b0;
                clk_counter <= clk_counter + 1;
            end
            3'b100 : begin
                SPI_CLK  <= 1'b1;
                rx_temp_reg[bit_counter] <= SPI_MISO;
                clk_counter <= clk_counter +1;
            end
            3'b111 : begin
                if(bit_counter != 3'b000) begin
                    bit_counter <= bit_counter - 1;
                    clk_counter <= 3'b000;
                end else begin
                    bit_counter <= 3'b111;
                    clk_counter <= 3'b000;
                    rx_FIFO[rx_addrw] <= rx_temp_reg;
                    rx_addrw <= rx_addrw + 1;
                    if(command_empty == 1'b1) cur_state <= SPIED;
                    else begin
                        if(command_FIFO[command_addrr] == 2'b01) cur_state <= SPITX;
                        else cur_state <= SPIRX;
```

```
                            command_addrr <= command_addrr + 1;
                        end
                    end
                end
                default : begin
                    clk_counter <= clk_counter + 1;
                end
            endcase
        end
        SPIED : begin
            case(clk_counter)
                3'b000 : begin
                    SPI_EN  <=1'b1;
                    SPI_CLK <=1'b0;
                    clk_counter <= clk_counter + 1;
                end
                3'b100 : begin
                    SPI_EN   <= 1'b0;
                    SPI_CLK  <= 1'b0;
                    SPI_MOSI <= 1'b0;
                    clk_counter <= 3'b000;
                    cur_state <= IDLE;
                end
                default : begin
                    clk_counter <= clk_counter + 1;
                end
            endcase
        end
    endcase
end


endmodule
```

SPI_controller.v:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2024/09/22 14:34:42
// Design Name:
// Module Name: TS_controller
// Project Name:
// Target Devices:
// Tool Versions:
```

```
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module SPI_controller(
    input clk,

    input wire [31:0] PC_rx,
    input wire [31:0] PC_addr,
    input wire [31:0] PC_val,
    output reg [31:0] PC_tx,
    output reg command_read,
    output reg rx_read,
    output reg tx_read,
    output reg [7:0] tx_byte,
    output reg [1:0] rw,
    output reg [9:0] cur_state,
    input wire [7:0] rx_byte,
    input wire busy
    );

    reg busy_reg = 0;
    reg [7:0] tx_byte_reg;
    reg [7:0] rx_byte_reg;
    reg [2:0] read_counter;
    reg [31:0] PC_rx_reg1, PC_rx_reg2;

    localparam idle_     = 10'b0000000001;
    localparam start_wr  = 10'b0000000010;
    localparam tx_wr1    = 10'b0000000100;
    localparam tx_wr2    = 10'b0000001000;
    localparam end_wr    = 10'b0000010000;
    localparam start_rt  = 10'b0000100000;
    localparam tx_rt     = 10'b0001000000;
    localparam rx_rt     = 10'b0010000000;
    localparam wait_rt   = 10'b0100000000;
    localparam end_rt    = 10'b1000000000;
```

```
initial begin
    cur_state <= idle_;
    PC_rx_reg1 <= 0;
    PC_rx_reg2 <= 0;
    tx_byte_reg <= 0;
    rx_byte_reg <= 0;
    busy_reg <= 1'b0;
    read_counter <= 0;
end


integer i;
```

```
always @(posedge clk) begin
    for (i=0; i<8; i=i+1) begin
        tx_byte[i] <= tx_byte_reg[i];
        rx_byte_reg[i] <= rx_byte[i];
    end
end
```

```
always @(posedge clk) begin
    case (cur_state)
        idle_ : begin
            command_read <= 1'b0;
            tx_read <= 1'b0;
            rx_read <= 1'b0;
            PC_rx_reg1 <= PC_rx;
            PC_rx_reg2 <= PC_rx_reg1;
            if (PC_rx_reg2[0] == 1'b0 && PC_rx_reg1[0] == 1'b1) begin
                cur_state <= start_wr;
            end
            if (PC_rx_reg2[1] == 1'b0 && PC_rx_reg1[1] == 1'b1) begin
                cur_state <= start_rt;
            end
        end
        //Write single byte
        start_wr: begin
            tx_byte_reg <= {1'b1, PC_addr[6:0]};
            cur_state <= tx_wr1;
        end
        tx_wr1: begin
            tx_byte_reg <= PC_val[7:0];
            command_read <= 1'b1;
            rw <= 2'b01;
```

```
            tx_read <= 1'b1;
            cur_state <= tx_wr2;
        end
        tx_wr2: begin
            command_read <= 1'b1;
            rw <= 2'b01;
            tx_read <= 1'b1;
            cur_state <= end_wr;
        end
        end_wr : begin
            tx_byte_reg <= {8{1'b0}};
            command_read <= 1'b0;
            tx_read <= 1'b0;
            cur_state <= idle_;
        end
        //Read two byte
        start_rt: begin
            tx_byte_reg <= {1'b0, PC_addr[6:0]};
            cur_state <= tx_rt;
        end
        tx_rt: begin
            command_read <= 1'b1;
            rw <= 2'b01;
            tx_read <= 1'b1;
            cur_state <= rx_rt;
        end
        rx_rt : begin
            command_read <= 1'b1;
            tx_read <= 1'b0;
            rw <= 2'b10;
            cur_state <= wait_rt;
        end
        wait_rt : begin
            tx_byte_reg <= {8{1'b0}};
            command_read <= 1'b0;
            rw <= 2'b00;
            busy_reg <= busy;
            if (busy_reg == 1'b1 && busy == 1'b0) begin
                rx_read <= 1'b1;
                cur_state <= end_rt;
            end
        end
        end_rt: begin
            rx_read <= 1'b0;
```

```
                read_counter <= read_counter + 1;
                if (read_counter == 2) begin
                    PC_tx[7:0] <= rx_byte_reg;
                    read_counter <= 0;
                    cur_state <= idle_;
                end
            end
            default : begin
                tx_byte_reg <= {8{1'b0}};
                cur_state <= idle_;
            end
        endcase
    end



endmodule
```

USB_Driver.v:

```
`timescale 1 ps / 1 ps


module USB_Driver(
    input clk,
```

```
    input   wire    [4:0] okUH,
    output  wire    [2:0] okHU,
    inout   wire    [31:0] okUHU,
    inout   wire    okAA,

    output [31:0]PC_rx,
    input  [31:0]PC_tx,
    output [31:0]PC_command,
    output [31:0]PC_addr,
    output [31:0]PC_val,

    input   FIFO_wr_clk,
    input   FIFO_read_reset,
    input   FIFO_write_reset,
    input   FIFO_wr_enable,
    input   [31:0]FIFO_data_in,
    output  FIFO_full, // currently not used
    output  FIFO_BT,
    output FIFO_read_enable,
    input USB_ready
```

```
);

wire okClk;            //These are FrontPanel wires needed to IO communication
wire [112:0]    okHE;  //These are FrontPanel wires needed to IO communication
wire [64:0]     okEH;  //These are FrontPanel wires needed to IO communication
//This is the OK host that allows data to be sent or recived
okHost hostIF (
    .okUH(okUH),
    .okHU(okHU),
    .okUHU(okUHU),
    .okClk(okClk),
    .okAA(okAA),
    .okHE(okHE),
    .okEH(okEH)
);

//Depending on the number of outgoing endpoints, adjust endPt_count accordingly.
//In this example, we have 1 output endpoints, hence endPt_count = 1.
localparam  endPt_count = 2;
wire [endPt_count*65-1:0] okEHx;
okWireOR # (.N(endPt_count)) wireOR (okEH, okEHx);

//Wire In//////////////////////////
okWireIn wire10 (   .okHE(okHE),
                    .ep_addr(8'h00),
                    .ep_dataout(PC_rx));
okWireIn wire11 (   .okHE(okHE),
                    .ep_addr(8'h01),
                    .ep_dataout(PC_command));
okWireIn wire12 (   .okHE(okHE),
                    .ep_addr(8'h02),
                    .ep_dataout(PC_addr));
okWireIn wire13 (   .okHE(okHE),
                    .ep_addr(8'h03),
                    .ep_dataout(PC_val));



//Wire Out/////////////////////////////////
okWireOut wire20 (  .okHE(okHE),
                    .okEH(okEHx[ 0*65 +: 65 ]),
                    .ep_addr(8'h20),
                    .ep_datain(PC_tx));

wire [31:0] FIFO_data_out;
```

```
    wire prog_full;
    fifo_generator_0 FIFO_for_Counter_BTPipe_Interface (
        .wr_clk(FIFO_wr_clk),
        .wr_rst(FIFO_write_reset),
        .rd_clk(okClk),
        .rd_rst(FIFO_read_reset),
        .din(FIFO_data_in[9:2]),
        .wr_en(FIFO_wr_enable),
        .rd_en(FIFO_read_enable),
        .dout(FIFO_data_out),
        .full(FIFO_full),
        .prog_full(prog_full),
        .empty(FIFO_empty)
    );

    okBTPipeOut CounterToPC (
        .okHE(okHE),
        .okEH(okEHx[ 1*65 +: 65 ]),
        .ep_addr(8'ha0),
        .ep_datain(FIFO_data_out),
        .ep_read(FIFO_read_enable),
        .ep_blockstrobe(FIFO_BT),
        .ep_ready(prog_full)
    );
endmodule
```

CVM300_driver.v:

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date: 2024/11/03 11:05:11
// Design Name:
// Module Name: CVM300_driver
// Project Name:
// Target Devices:
// Tool Versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
```

```
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////


module CVM300_driver(
    input clk,

    input  CVM300_CLK_OUT,
    output CVM300_CLK_IN,
    output CVM300_SYS_RES_N,
    output CVM300_FRAME_REQ,
    output CVM300_SPI_EN,
    output CVM300_SPI_CLK,
    input  CVM300_SPI_OUT,
    output CVM300_SPI_IN,
    input  CVM300_LVAL,
    input  CVM300_DVAL,
    input [9:0] CVM300_D,

    input  [31:0]PC_rx,
    output [31:0]PC_tx,
    input  [31:0]PC_command,
    input  [31:0]PC_addr,
    input  [31:0]PC_val,

    output  FIFO_wr_clk,
    output  FIFO_read_reset,
    output  FIFO_write_reset,
    output  FIFO_wr_enable,
    output  [31:0]FIFO_data_in,
    input   FIFO_full, // currently not used
    input   FIFO_BT,

    output USB_ready
    );
```

```
    wire [3:0] SPI_state;
    wire busy;
    wire command_read;
    wire rx_read;
    wire tx_read;
    wire [1:0] Spi_rw;
    wire [7:0] Spi_rx_reg;
```

```
wire [7:0] Spi_tx_reg;
wire [9:0] controller_state;
//SPI SERDES
SPI_driver SPI_driver(
.clk(clk),
.cur_state(SPI_state),

.SPI_MISO(CVM300_SPI_OUT),
.SPI_MOSI(CVM300_SPI_IN),
.SPI_CLK(CVM300_SPI_CLK),
.SPI_EN(CVM300_SPI_EN),

.busy(busy),
.command_read(command_read),
.rx_read(rx_read),
.tx_read(tx_read),
.Spi_rw(Spi_rw),
.Spi_rx_reg(Spi_rx_reg),
.Spi_tx_reg(Spi_tx_reg)
);
//SPI controller
SPI_controller SPI_controller(
.clk(clk),
.PC_rx(PC_rx),
.PC_addr(PC_addr),
.PC_val(PC_val),
.PC_tx(PC_tx),
.command_read(command_read),
.rx_read(rx_read),
.tx_read(tx_read),
.rw(Spi_rw),
.tx_byte(Spi_tx_reg),
.rx_byte(Spi_rx_reg),
.busy(busy),
.cur_state(controller_state)
);
```

```
// Reset control////////////////////////////////////////////////////////////////

// Clock generation////////////////////////////////////////////////////////////////
reg CVM_Clk;
```

```
reg [23:0] ClkDivCVM = 24'd0;
assign CVM300_CLK_IN = CVM_Clk;
```

```
always @(posedge clk) begin
    if (ClkDivCVM == 4) begin
        CVM_Clk <= !CVM_Clk;
        ClkDivCVM <= 0;
    end else begin
        ClkDivCVM <= ClkDivCVM + 1'b1;
    end
end

// Frame request && Reset set//////////////////////
reg reset = 1;
reg started = 0;
reg frame_request = 0;
reg [15:0] HS_counter = 0;
always @(posedge CVM_Clk) begin
    if (PC_command[0] == 1) begin
        reset <= 1;
        started <= 1;
        HS_counter <= 16'd1;
    end
    else begin
        if (started) reset <= 0;
        else reset <= 1;
    end

    if (PC_command[16:1] == HS_counter && started == 1'b1) begin
        HS_counter <= HS_counter + 1;
        frame_request <= 1'b1;
    end
    else begin
        frame_request <= 1'b0;
    end

end
assign CVM300_SYS_RES_N = ~reset;
assign CVM300_FRAME_REQ = frame_request;
assign FIFO_read_reset = frame_request;
assign FIFO_write_reset = frame_request;
reg[31:0] FIFO_data_in_reg;
reg FIFO_ready_reg;
reg FIFO_wrena_reg;
reg read_flag=1'b0;
reg CMV_clk_CDC;
```

```
    reg CMV_clk;
    reg CMV_clk_reg;

    always @(posedge clk) begin
        CMV_clk_CDC <= CVM300_CLK_OUT;
        CMV_clk <= CMV_clk_CDC;
        CMV_clk_reg <= CMV_clk;
        if (CMV_clk ==1'b0 && CMV_clk_reg == 1'b1) begin
            if (CVM300_LVAL == 1'b1 && CVM300_DVAL == 1'b1) begin
                FIFO_data_in_reg[9:0] <= CVM300_D;
                FIFO_data_in_reg[31:10] <= 0;
                read_flag <= 1'b1;
                FIFO_wrena_reg <= 1'b1;
            end
        end
        else FIFO_wrena_reg <= 1'b0;
        if (CVM300_LVAL == 1'b0 && CVM300_DVAL == 1'b0) begin
            if (read_flag == 1'b1) FIFO_ready_reg <= 1'b1;
            read_flag <= 1'b0;
        end
        if (FIFO_BT == 1'b1) FIFO_ready_reg <= 1'b0;
    end

    assign FIFO_data_in = FIFO_data_in_reg;
    assign FIFO_wr_enable = FIFO_wrena_reg;
    assign FIFO_wr_clk = clk;
    assign USB_ready = FIFO_ready_reg;



endmodule
```

Python code:

Lab9.py:

```
# -*- coding: utf-8 -*-


#%%
# import various libraries necessary to run your Python code
import time   # time related library
import sys,os    # system related library
import numpy as np
import matplotlib.pyplot as plt
ok_sdk_loc = "C:\\Program Files\\Opal Kelly\\FrontPanelUSB\\API\\Python\\x64"
ok_dll_loc = "C:\\Program Files\\Opal Kelly\\FrontPanelUSB\\API\\lib\\x64"
```

```
sys.path.append(ok_sdk_loc)   # add the path of the OK library
os.add_dll_directory(ok_dll_loc)
```

```
import ok     # OpalKelly library
#%%
def reset_image_sensor():
    dev.SetWireInValue(0x01, 1)
    dev.UpdateWireIns()
    time.sleep(0.5)
    dev.SetWireInValue(0x01, 0)
    dev.UpdateWireIns()
    time.sleep(0.1)
#%%
def write_to_device(reg_addr, value):
    dev.SetWireInValue(0x00, 0)
    dev.UpdateWireIns()
    dev.SetWireInValue(0x02, reg_addr)
    dev.SetWireInValue(0x03, value)
    dev.UpdateWireIns()  # Update the WireIns
    time.sleep(0.1)
    dev.SetWireInValue(0x00, 1) # Write trigger
    dev.UpdateWireIns()  # Update the WireIns
    time.sleep(0.1)
    dev.SetWireInValue(0x00, 0)
    dev.UpdateWireIns()  # Update the WireIns


#%%
def read_from_device(reg_addr):
    dev.SetWireInValue(0x00, 0)
    dev.UpdateWireIns()  # Update the WireIns
    time.sleep(0.1)
    dev.SetWireInValue(0x02, reg_addr)
    dev.SetWireInValue(0x00, 2)  # Read trigger
    dev.UpdateWireIns()  # Update the WireIns
    time.sleep(0.1)
    dev.UpdateWireOuts()
    read = dev.GetWireOutValue(0x20)
#    if slave_addr == 0x3C:
#        m_L = read // 2**8
#        m_H = read - (m_L * 2**8)
#        read =  m_H * 2**8 + m_L
#    if read >= 2**15:
#        read = read - 2**16 # deal with 2's complement
    dev.SetWireInValue(0x00, 0)
```

```
    dev.UpdateWireIns()
    return read
#%%
def setup_image_sensor():
    print("setting up...")
    reset_image_sensor()
    write_to_device(3, 8)
    write_to_device(4, 160)
    write_to_device(57, 3)
    write_to_device(58, 44)
    write_to_device(59, 240)
    write_to_device(60, 10)
    write_to_device(69, 9)
    write_to_device(80, 2)
    write_to_device(83, 187)
    write_to_device(97, 240)
    write_to_device(98, 10)
    write_to_device(100, 112)
    write_to_device(101, 98)
    write_to_device(102, 34)
    write_to_device(103, 64)
    write_to_device(106, 94)
    write_to_device(107, 110)
    write_to_device(108, 91)
    write_to_device(109, 82)
    write_to_device(110, 80)
    write_to_device(117, 91)
    print("setting up done")
#%%
def read_a_frame(HS_counter):
    buf = bytearray(315392)
    dev.SetWireInValue(0x01, HS_counter)
    dev.UpdateWireIns()
    dev.ReadFromBlockPipeOut(0xa0, 1024, buf)
    return buf
#%%
# Define FrontPanel device variable, open USB communication and
# load the bit file in the FPGA
dev = ok.okCFrontPanel()  # define a device for FrontPanel communication
SerialStatus=dev.OpenBySerial("")      # open USB communication with the OK board
# We will NOT load the bit file because it will be loaded using JTAG interface from Vivado
```

```
# Check if FrontPanel is initialized correctly and if the bit file is loaded.
# Otherwise terminate the program
```

```
print("------------------------------------------------------")
if SerialStatus == 0:
    print ("FrontPanel host interface was successfully initialized.")
else:
    print ("FrontPanel host interface not detected. The error code number is:" + str(int(SerialStatus)))
    print("Exiting the program.")
    sys.exit ()
```

```
#%% Reg and value constants
HS_counter = 0
#%%
# Define the two variables that will send data to the FPGA
# We will use WireIn instructions to send data to the FPGA
time.sleep(1)
setup_image_sensor()
while (True):
    input()
    HS_counter = HS_counter + 2
    buf = read_a_frame(HS_counter)
    width, height = 648, 480
    arr = np.frombuffer(buf, dtype=np.uint8, count=314928)
    arr = arr.reshape(486, 648)
    plt.imshow(arr, cmap = 'gray')
    plt.show()
#  # Read data from BT PipeOut
```

```
#for i in range (0, 1024, 1):
#   result = buf[i];
#   print (result)
```

```
dev.Close
```