GH repo: https://github.com/evgerritz/cpsc429_labs/tree/2_p2
Steps to run:
On Zoo:

1. Boot server vm and forward the VNC and server ports

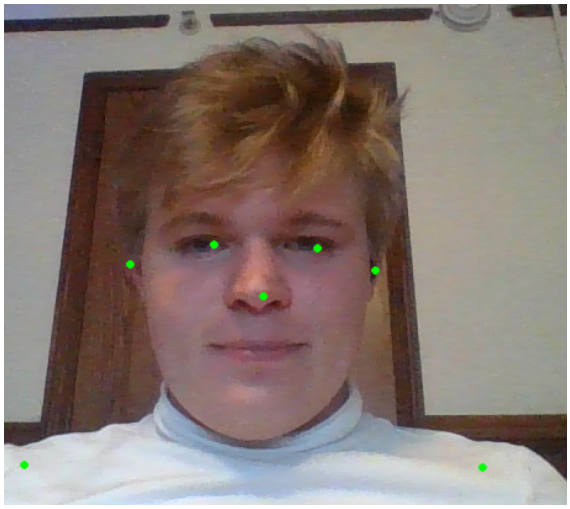
On Host:

1. Use VNC to connect
2. Enter movenet_server/
3. cargo run

On Target:

1. Connect camera to VM
2. Enter rust_movenet/
3. ./tunnel
4. cargo run

Part 1:

- Acknowledgments
  - Used the provided resources
- Challenges
  - None
- 

Part 2:

- Acknowledgments
    - Used the provided resources
    - https://riptutorial.com/rust/example/4404/a-simple-tcp-client-and-server-application--echo
- Challenges
    - Getting all of the forwarding to work in order to connect to the VM on the Zoo was tricky.
    - Ultimately, I could not get kvm acceleration to work on the zoo (the zoo uses qemu version 2.3), which made it impossible to run the rust movenet server (tflitec had not finished building after many hours). I was however, able to get a basic server connected and successfully sent and received data from that. The rest of the assignment, however, was done using a server and client on the same virtual machine.
    - Coordinating the reads and writes between the server and the client was also difficult until I switched to using a BufReader. This allowed me to use the read_exact method, which gave correct results from the server.
- Overview/Discussion
    - I chose to divide the server and the client's tasks by having the client send and receive as bytes the input and output, respectively, of the network running on the server. 'client.rs' contains the code for connecting to and sending bytes to/from the server, as well as some functions for converting to/from bytes. The movenet_server cargo project contains the code for the server-side DNN interpreter. It accepts TCP streams and then repeatedly reads input from the TCP stream, feeds that into the network, and then writes the output of the network back to the TCP stream.
    - For correctness, I ensured that the resulting image matched the original results when. I also used read_exact to guarantee that each frame would be read individually and at once by the server. Initially, my results were incorrect because of incomplete reads.
    - For performance, I calculated the number of images the system was capable of processing per second and got an average value of 7.6 images processed per second.
    - For the app interface, I simplified interactions with the network by creating a Server struct that contains a TcpStream. The client can send or receive bytes to the server through corresponding methods. I also wrote some functions to simplify the client's conversion of data to and from bytes. From the client's perspective, all it needs to do is convert the data to bytes, send the data to the

server, receive and convert the output from the server, and display the result. To ensure good networking performance, I decided to send each frame at once to the server instead of splitting it into smaller chunks. Since the data can always fit into two TCP packets, it is not likely not too big for chunks of this size to be an issue.

GH repo: https://github.com/evgerritz/cpsc429_labs/tree/2_p3
Steps to run:
On Zoo:

1. Boot server vm and forward the VNC and server ports

On Host:

1. Use VNC to connect to server
2. Enter movenet_server/
3. cargo run

On Target:

5. Connect camera to VM
6. Start yuv422 webcam with ./fake_webcam
7. Enter rust_movenet/
8. ./tunnel (to connect to the Zoo)
9. cargo run

Part 3:

- Acknowledgements
    - Resources provided in assignment
    - Mmap streaming example: https://www.kernel.org/doc/html/v4.9/media/uapi/v4l/mmap.html
    - Videodev2 header: https://www.kernel.org/doc/html/v5.7/media/uapi/v4l/videodev.html
    - Another v4l example: https://medium.com/@athul929/capture-an-image-using-v4l2-api-5b6022d79e1d
    - Setting up virtual yuv422 webcam: https://stackoverflow.com/questions/59574987/how-to-change-mjpeg-to-yuyv422-from-a-webcam-to-a-v4l2loopback
    - Code I translated into rust for converting from yuv422 to rgb: https://github.com/kd40629rtlrtl/yuv422_to_rgb/blob/master/yuv_to_rgb.c
- Challenges
    - Getting the virtual webcam set up
    - Creating Rust structs that were compatible with the ioctl calls was difficult. I had to use gdb to print the physical memory layout of the v4l2_buffer struct in order to get the alignment right.
    - Debugging failed ioctl calls
    - Converting between libc::c_void and &[u8]

    - Reading a stream of RGB bytes into an OpenCV Mat
    - Debugging when images were incorrectly converted
- Overview
    - The majority of the code for this assignment is in rust_movenet/v4l_utils.rs.
        - v4l_utils.rs implements the Rust interface for v4l, as will be used by the client to capture images from the webcam.
        - main.rs has been rewritten to use the v4l wrappers in v4l_utils.rs.
    - movenet_server/main.rs now includes a yuv422-to-rgb24 conversion function, and the resize_with_padding function from rust_movenet/util.rs.
    - The general workflow is as follows:
        - The client uses v4l to capture a yuv422 buffer, and sends it as bytes to the server.
        - The server then converts that yuv422 buffer to rgb24, creates an OpenCV matrix, and then does the required resizing of input for the network.
        - The server then runs the DNN interpreter, and sends the results as bytes back to the client.
        - The client converts these bytes to floats, draws the output on a blank frame, and then displays the frame.

GH repo: https://github.com/evgerritz/cpsc429_labs/tree/2_p4
Steps to run:
On Zoo:
Boot server vm (./vms/boot); this script automatically forwards the VNC and server ports

On Host:
Use VNC to connect to server
Enter movenet_server/
cargo run

On Target:
Connect camera to VM (Checkbox in Devices menu for VirtualBox)
Enter rust_movenet/
Start yuv422 webcam with ./fake_webcam
./tunnel (to connect to the Zoo)
./test_module.sh (gets latest version of rust_camera.ko and inserts into kernel)
./run (starts rust_movenet as root)

Part 4:

- Acknowledgments
    - Resources provided in assignment
    - http://fivelinesofcode.blogspot.com/2014/03/how-to-translate-virtual-to-physical.html
    - https://elixir.bootlin.com/linux/latest/source/arch/csky/include/asm/page.h#L38
    - https://www.unix.com/man-page/suse/9/filp_open
- Main files for this assignment
    - lab2/part4/rust_kmod/custom_modules/rust_camera.rs
    - lab2/part4/rust_movenet/src/main.rs
- Challenges
    - Getting the pfns from pagemap. I didn't know we had to execute the program with root privileges so I spent a while debugging these pfns to no avail.
    - Getting the vfs_ioctl calls to work. These were hard to debug and the lack of the usual Rust memory checking forced me to think through when each object was valid and how the kernel module and userspace program's executions overlapped.

    - Using pointers in Rust was always tricky. Additionally, often there was a memory issue, it caused a segfault in the kernel, and I had to reboot the virtual machine.
    - Setting up the TCP stream. I had to read through a lot of the bindings code as well as the relevant Linux man pages to understand how to use the networking code. I also had to change the visibility of the TcpStream struct's sock member in order to instantiate it directly.
- Overview
    - In userspace, we call mmap to get the userspace virtual addresses for the video buffer, use /proc/self/pagemap to get the corresponding page frame numbers, and then send them as well as our userspace buffer's address to the kernel module.
    - The kernel module then spawns a thread to queue and dequeue buffers for v4l, sending the buffers (accessed through their kernel address) to the server. The server computes the output of the network, sends it back to the kernel module, and the result is stored in a global buffer protected by a mutex.
    - The userspace program periodically obtains the contents of the kernel module's output buffer and displays the output for the user.