Introduction

We live in a world where there is an increase in demand for efficient computer vision pipelines that can support multiple cameras. The computer vision pipelines that are in use right now run mostly but not completely on the GPU; data is passed back and forth between CPU and GPU for different stages of the pipeline. We aim to try to move more of the pipeline into the GPU, to avoid extra data transfers, and have mostly only control moving between CPU and GPU.

System Architecture

The computer vision pipeline that we worked on was a simple detection and tracking pipeline, implemented with the *darknet* [1] [2] neural network framework. It has 6 stages: capture, preprocess, detect, track, draw, and output.

The capture stage was a simple call to OpenCV's VideoCapture [3] module. The preprocessing stage consisted of subsampling (to 224x224), color format conversion (24-bit RGB to 24-bit BGR), and converting from OpenCV's Mat object to darknet's `image_t` object type. The detection stage was a forward-pass through the yolov3 [4] neural network implemented in darknet, where the input is the 224x224 24-bit BGR image, and the output is a list of bounding boxes. The detection stage includes some post-processing as well, to filter the bounding boxes, and to convert the raw neural network output to actual bounding boxes. The tracking stage assigned tracking IDs, by looking sequentially at the past 1 second worth of frames, and matching tracking IDs to the new frame by centroid distance. The drawing stage is simply drawing all the bounding boxes, including their coordinates, confidence, and label, onto the frame. Finally, the output stage is either simply printing the list of detections and tracking IDs, saving all the final frames to a video, or showing the frames (this is determined by a run-time parameter).

Since the final fps of the frames shown live were heavily dependent on the client machine's screen size (eg.

[1] https://github.com/pjreddie/darknet
[2] https://github.com/AlexeyAB/darknet
[3] https://docs.opencv.org/4.3.0/d8/dfe/classcv_1_1VideoCapture.html
[4] https://pjreddie.com/darknet/yolo

viewing the stream on a 23" monitor achieved around 10 fps, while on a 10" monitor achieved close to 20 fps), we decided to take all measurements with no output at all, and only use the stream output and video output for validation that the whole pipeline works, and the printing output to validate the detections and tracking.

Baseline

In our baseline model [5] (Figure 1), which was based on a demo program [6] included in darknet, all stages except for the detection stage were executed completely on the CPU. We had thread-safe inter-stage queues (implemented as a wrapper around C++ `std::deque`), and every stage ran in a loop in its own thread; dequeue from the inter-stage queue, execute the stage, enqueue into the next inter-stage queue. The contents of each stage are as described above. Besides the detection stage, all stages as described above executed completely on the CPU.

The detection stage is a forward-pass through the neural network, with some post-processing at the end. At the program level, the control of the forward-pass through the neural network ran on the CPU, looping through every layer of the neural network and executing the forward-pass of the layer on the GPU. The data stayed in the GPU throughout all forward-passes, of course with a memory copy into and out from the GPU before and after the whole network, respectively. The post-processing was done completely in the CPU, however.

[5] https://github.com/Rahi374/darknet/tree/baseline
[6] https://github.com/AlexeyAB/darknet/blob/master/src/yolo_console_dll.cpp

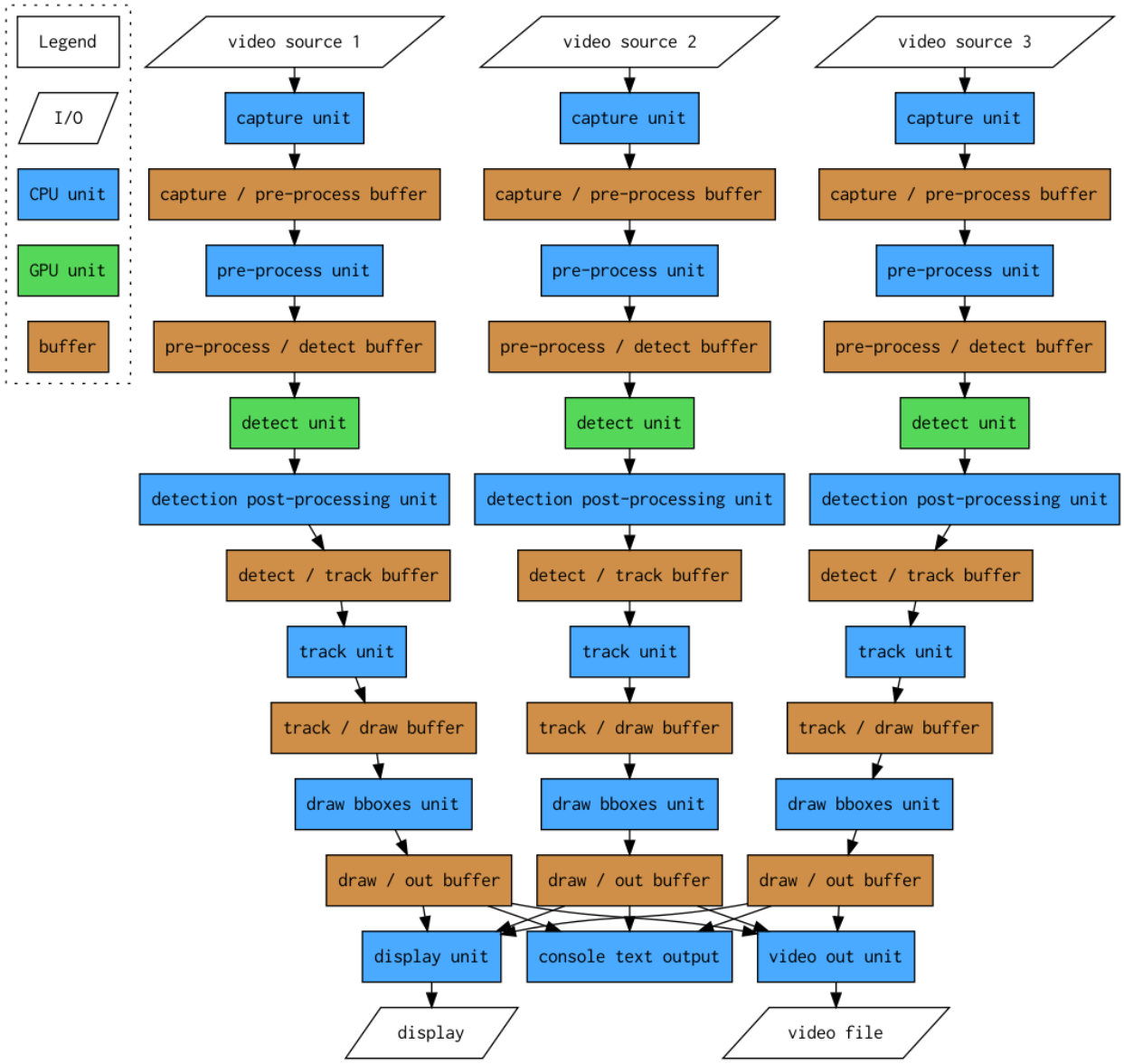

Figure 1: Baseline model (with 3 concurrent video sources)

Our model

With our model [7] (Figure 2), we aim to reduce the number of data transfers between the CPU and GPU, while moving more of the processing into the GPU. For the latter, we moved the preprocessing stage and the tracking stage to the GPU. The preprocessing stage was quite simple to parallelize and implement in the GPU. The tracking stage was more difficult, and will be discussed later. For reducing the amount of data transfers, we extended the inter-stage queues to contain a pointer to GPU memory, and a GPU event. That way, for example, after the the CPU issues the preprocessing stage GPU kernel, it can enqueue the completion event and preprocessing output GPU memory pointer into the inter-stage queue, and the detection stage can wait on the completion event, and simply feed the GPU memory pointer with the preprocessing output to the input of the detection GPU kernel. This removes one pair of device-to-host and host-to-device memory copies.

Due to the time constraint, however, we were unable to move the post-processing part of the detection stage into the GPU. In the Future work section we will describe how one would go about implementing it with our code.

To recap, what we contributed on top of the baseline was to move the preprocessing stage and the tracking stage to the GPU, and we removed the CPU-GPU memory copy between the preprocessing stage and the detection stage with GPU memory pointer and GPU event in the inter-stage queue.

[7] https://github.com/Rahi374/darknet

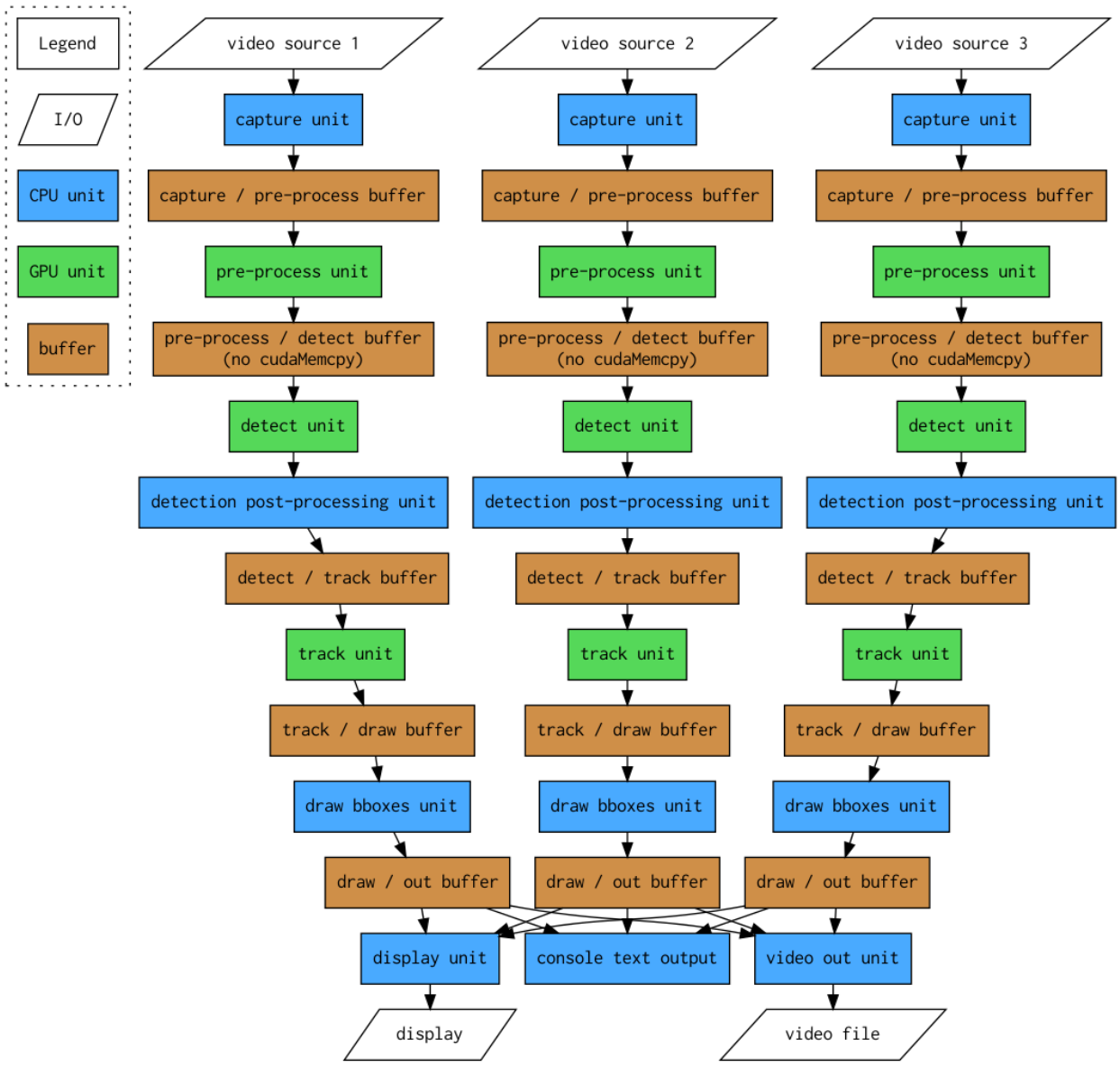

Figure 2: New model (with 3 concurrent video sources)

Parallel tracking algorithm

The tracking algorithm assigns tracking IDs to the new detections, based on 1 second worth of old detections. We go backwards sequentially through the old detections, matching whatever detections we can in order.

- Input: list of new detections without tracking IDs, list of lists of old detections with tracking IDs
- Output: list of new detections with tracking IDs

We do work for some old list of detections D. Let the list of new detections be N. We allocate a $|D| \times |N|$ working space, and have $|D| \times |N|$ nodes. Let i be the row number such that $D = \{d_0, d_1, ...d_i, ...d_{|D|}\}$, and j be the column number such that $N = \{n_0, n_1, ..., n_j, ...n_{|N|}\}$.

In parallel, every node (i, j) finds the centroid distance between the detection j from D and detection i from N. If the object class doesn't match, the distance is over the threshold, or detection i already has a tracking ID, then an infinite distance is recorded. If the detection i has a tracking ID that matches detection j's tracking ID, then it means that detection j has already been matched with a new detection, so we record a negative distance.

Next, in parallel, for every j, we do a parallel odd-even transposition sort. This means that for every old detection j, we will have a sorted list of new detections by their distance from detection j.

Finally, in parallel, for every j, we have one worker node that walks the sorted list of new detections. For every detection i in the sorted list of new detections whose distance is less than the threshold, we (node j) use CAS to "claim" the match between detection j and detection i. This works nicely because if and only if the tracking ID was zero, it means detection i's tracking ID was not yet assigned. Additionally, if we find a negative distance, we stop searching, because a negative entry means that detection j's tracking ID has already been assigned. The reason why we have to check the threshold in this stage again is to avoid floating point comparison errors with "infinity".

Measurements, results, and analysis

We ran measurements for both the baseline model and our new model, with 1, 2, 3, and 4 video streams. The measurements that we recorded were, for every thread id and frame number, the latency (time taken for the frame to go through the whole pipeline), the average detection fps, the average capture fps, and the time spent waiting in the queue for every stage and the time taken to complete every stage for all stages (capture, preprocess, detect, track, and draw).

The raw measurements can be seen in [8], and are in a format easily importable to matlab/octave. The column headers are (all time units in ms): thread id, frame id, latency, detection fps, capture fps, time taken for capture, time spent in preprocessing queue, time taken for preprocessing, time spent in detect queue, time taken for detect, time spent in tracking queue, time taken for track, time spent in drawing queue, time taken for draw. The file names denote which model (baseline or new) and how many video streams (1-4) the trace was captured of. All of these measurements were taken with the same video stream, MOT20-02 [9] from MOTChallenge.

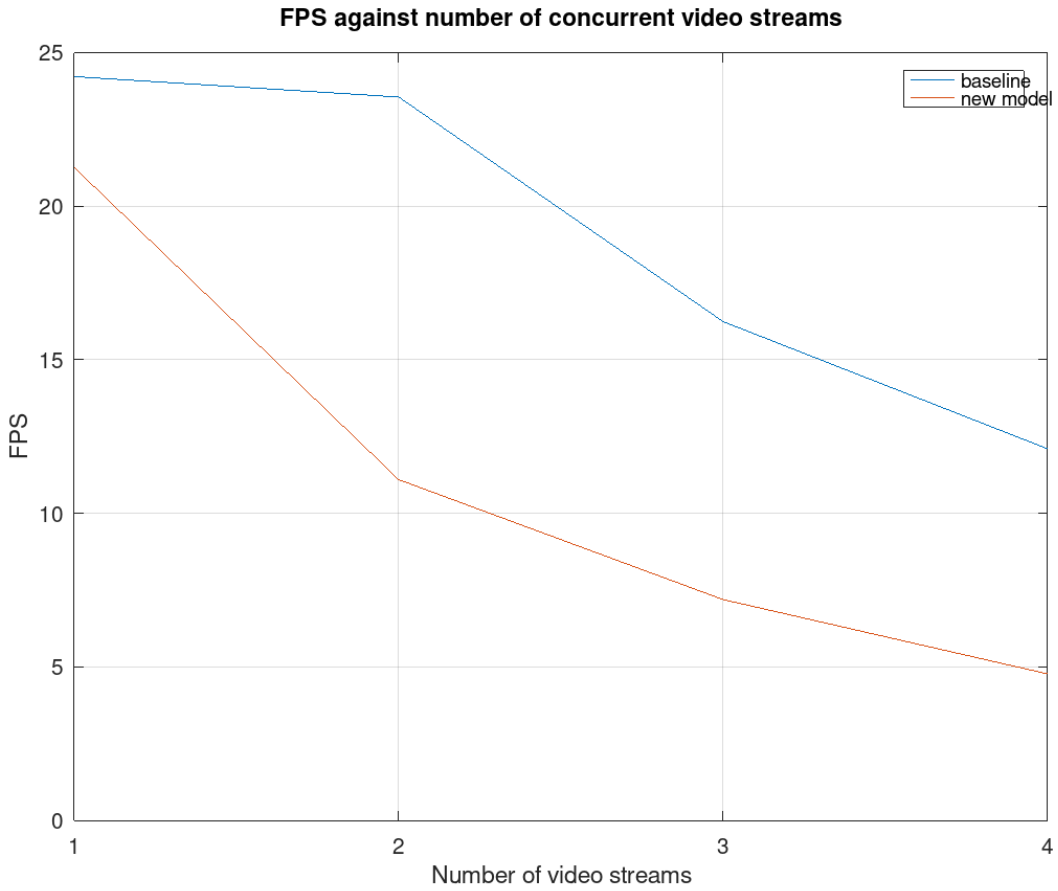

Figure 3: FPS against number of concurrent video streams

[8] https://github.com/Rahi374/cs-2910-gh/measurements/tree/master/measurements/octave
[9] https://motchallenge.net/data/MOT20

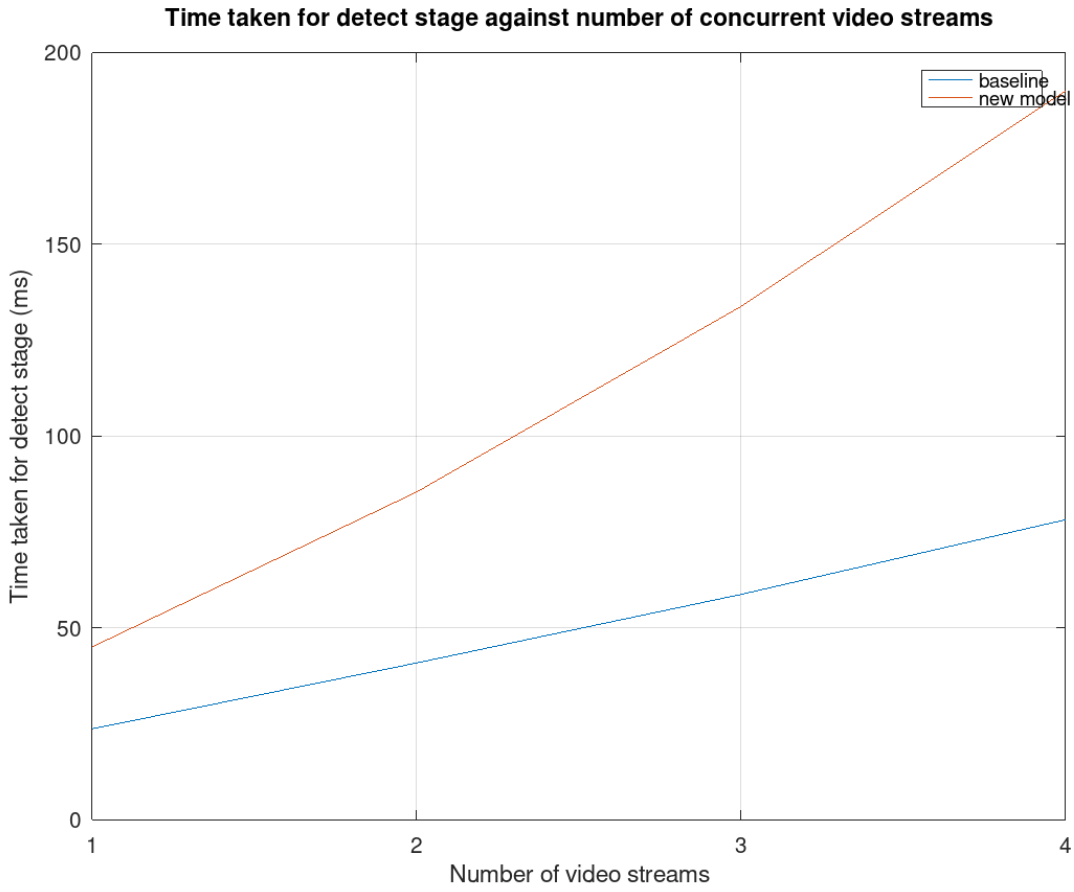

Figure 4: Time taken for detect stage against number of concurrent video streams

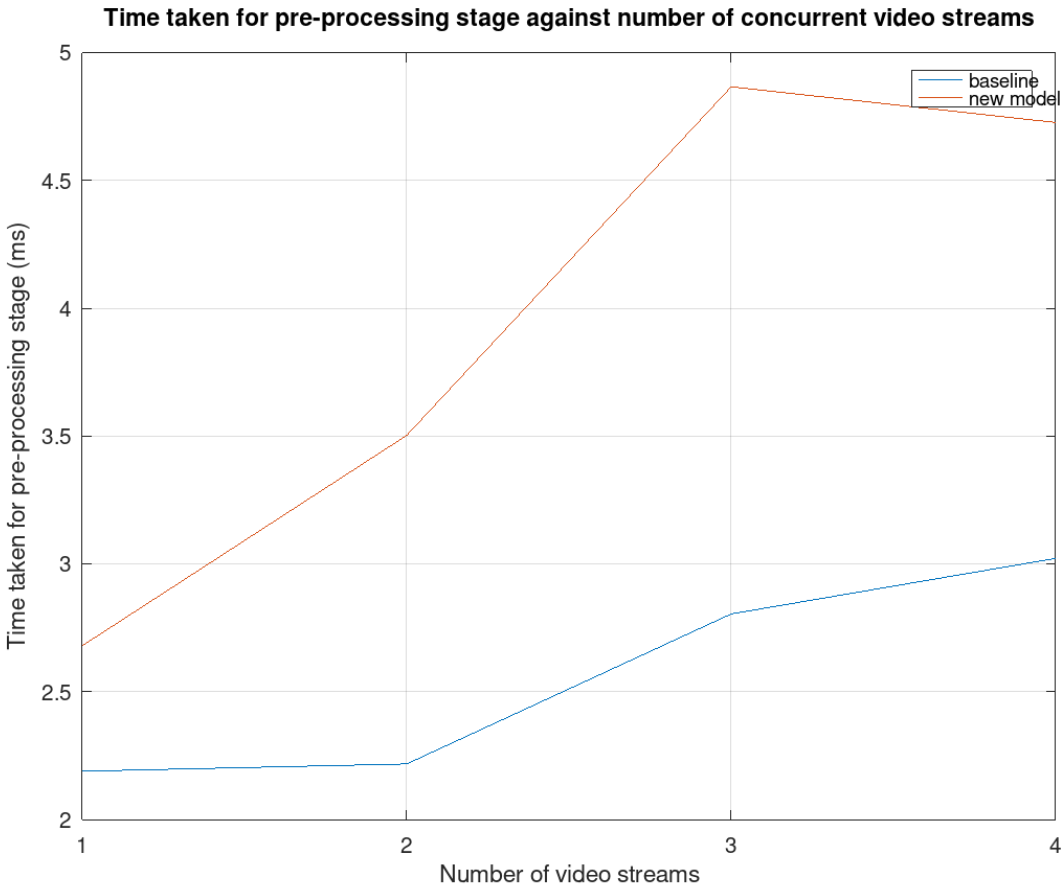

Figure 5: Time taken for pre-processing stage against number of concurrent video streams

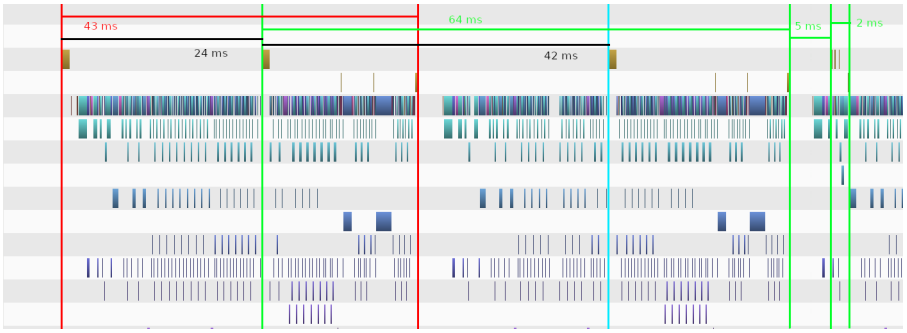

Figure 6: Snapshot of GPU execution timeline for new model [10]

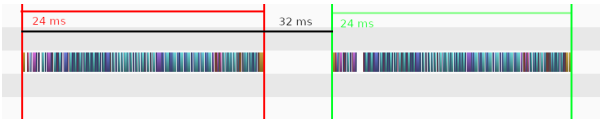

Figure 7: Snapshot of GPU execution timeline for baseline model [11]

We have extracted a few interesting values and graphed them in Figures 3, 4, and 5. Most notably, the fps for all number of concurrent video streams are lower in our new model compared the the baseline. There are a few causes of this drop. As we can see in the other two graphs, the time taken for the preprocessing stage and the detect stage are higher in the new model compared to the baseline model. These would certainly contribute to lower fps. The question is, why are the preprocessing and detection stages taking longer? The detection stage, at least, should be the same, since it is unchanged between the baseline model and the new model.

We take a look at Figures 6 and 7, which are snapshots from the Nvidia visual profiler (nvvp), for the execution of the first two frames in the stream for the new model and the baseline model, respectively, with only one video stream. The red time frame is the time for the first frame between entering and exiting the GPU, and the green time frame is for the second frame. For the baseline model, the only part of the pipeline that is in the GPU is the neural network part of the detection stage, so what we see in Figure 7 is that every frame consistently takes 24ms, and there are 32ms in between every frame. Since the source video is 25 fps, which is a frame interval of 40ms, we can see that the detection part of the pipeline is processed faster than the frame input rate, allowing us to have a high fps as indicated in Figure 3.

Now, in Figure 6, there is a bit more for the GPU to do for a given frame, since there is now preprocessing, detection, waiting for the cpu to do detection post-processing, and tracking, all on the GPU. Within the red time frame, the first yellow box is a memory copy, to bring in the raw video frame. This takes 1ms, since it is 1920x1080 pixels, multiplied by 3 color channels, which is 6.2MB, which is then copied over a 5GB/s link. After that is 90us of executing the preprocessing on the GPU. Then 24ms after the beginning of the memcpy for the preprocessing stage, the memcpy for the preprocessing stage of the next frame interrupts

[10] https://github.com/Rahi374/cs-2910-gh/blob/master/nvvp-logs/new-simple-1.nvvp
[11] https://github.com/Rahi374/cs-2910-gh/blob/master/nvvp-logs/baseline-simple-1.nvvp

the detection of the first frame. This obviously causes the detection of the first frame to take longer than in baseline.

What we were unable to figure out, however, is why the next frame would come in 24ms later. The memcpy and preprocessing stage only add a bit over 1ms, and in the baseline there are 32ms between the beginning of processing of each frame. One theory is that OpenCV's VideoCapture's frame capture free-runs, but that would mean that we should have a similar early frame 2 in the baseline timeline. If conversely OpenCV's VideoCapture captures at the framerate of the video, at 25fps, then we should have 40ms in between frames, which happens between frame 2 and frame 3 in the new model (albeit 42ms), but nowhere else.

The fact that in the new model there are multiple detection stages prempting each other explains why the detection stage is taking longer, but we were unable to figure out why the detection stages are executing at the timing that causes them to preempt each other. Originally the conclusion would have been that the additional time for the memcpy into the GPU preceeding the pre-processing stage caused the delay of the start of the detection stage to miss the 40ms deadline for processing one frame, but the fact that the pre-processing on GPU including the memcpy in is only 1~2ms, and that the interval between incoming frames is **not** 40ms, means that this conclusion doesn't quite work.

The green 5ms interval in Figure 6 is the time for the detection post-processing to run on the CPU, and the green 2ms interval is the time taken for the tracking stage to run on the GPU, including the memcpy in and out.

Future work

One way that the delay of the completion of the pipeline stages in the GPU (as well as the preemption) could be avoided is by having multiple pipeline units running in parallel. For example, while frame 1 is in the GPU in the detect stage, and frame 2 wants to come into the GPU as well, we could run another preprocessing and detection kernel on the GPU in a different stream in parallel. If there are available execution units, and if the GPU kernels are dispatched on different streams, then the kernels should run in parallel, allowing us to avoid the preemption and avoid missing the 40ms (for 25 fps) deadline. The difficulty in implementing

this is mainly due to the program architecture of darknet, since the detection stage is made up of multiple kernel calls, and they all determine the stream with the function call `get_cuda_stream()`, which automatically chooses the stream based on the device number. If there is only one GPU, then only one stream would be used. The best solution to this might be simply to allow specifying an integer parameter to `get_cuda_stream()`, to specify which stream the caller wants. All existing calls would have to be replaced with a parameter of zero, and then there needs to be some way to forward the stream number parameter from the Pipeline object all the way down to the layers of the neural network, where the actual GPU kernels are called. These multiple pipeline units could be allocated in many different ways, such as having two per pipeline per detection stage (with the exact number to be tuned), or to have multiple pipeline share pipeline units (which would need modifications to the Pipeline class and the startup).

Another improvement would be to move the detection stage post-processing to the GPU. This would remove the data transfer from GPU to CPU after the detection stage and the data transfer from CPU to GPU before the tracking stage. At a high level, the post-processing loops through the three YOLO layers in the neural network (`fill_network_boxes()` in `src/network.c`) and extracts their output (`get_yolo_detections()` in `src/yolo_layer.c`), does non-max suppression (`do_nms_sort()`), and converts them to `bbox_t` (`Detector::detect()` in `src/yolo_v2_class.cpp`). This last function is essentially the entry point to the whole detect stage. `get_yolo_detections` is where the main processing is. The GPU output that's stil on the GPU can be obtained from `l.output_gpu` (as opposed to `float *predictions = l.output`). `entry_index()` should be trivial, and the rest of the loop in `get_yolo_detections` seems to be embarrasingly parallelizable, as does `correct_yolo_boxes`. Finally, there needs to be some memory management to save the old detection vectors in GPU memory, so that future tracking stages can use them without having to memcpy and rearrage the vectors into a matrix.

Another improvement is to run the detection less frequently. Since the detection takes so long to run, perhaps we could run the detection every 20 frames, and for the other 19 frames run a tracking algorithm that does some amount of simple detection. For the purpose of the project, it would have to be implemented

in the GPU. One implementation that we considered was the correlation tracker implemented in dlib [12], however, it seemed to be implemented in the CPU, and would have required porting to GPU.

A minor improvement would be to use shared memory in the tracking stage. In our implementation of our parallel tracking algorithm, we do all operations in device memory; copying the vector of current detections to shared memory for each block would lead to some performance improvement. Since currently the tracking stage only takes 290us, this will likely not be a significant preformance boost, however.

Miscellaneous

To compile the libdarknet, as well as our model, simply run `make` in the project directory [13]. To run the program, run `./uselib`. Adding the `-h` parameter, as `./uselib -h` will give usage instructions. To build baseline, simply git checkout the baseline tag, `git checkout baseline`, and then compile and run as usual. Note that the current directory must be in the `LD_LIBRARY_PATH` environment variable, since `uselib` links to `libdarknet.so`, which is built in the project directory.

Due to a bug in OpenCV [14], some resolutions may not work. Notably, MOT20-03 will crash the program. The easiest solution is to simply re-encode it as a resolution that is known to work, such as 1920x1080.

This report and other information about this project can be found at [15].

[12] http://dlib.net/video_tracking_ex.cpp.html
[13] https://github.com/Rahi374/darknet
[14] https://github.com/opencv/opencv/issues/12466
[15] https://github.com/Rahi374/cs-2910-gh