AI-BASED CLOUD RENDERING: FULL-GPU PIPELINE IN FFMPEG

XIAOWEI WANG, NVIDIA DEVTECH

BACKGROUND

- AI-based rendering is becoming a trend in datacenter
  - The industry has already deployed applications
  - AI-based rendering is complicated, inference/graphics/computing/transcoding
  - A GPU rendering pipeline requires a lot of cooperation to implement, while there is no public reference solutions
- We want to build a reference design for AI-based cloud-rendering
  - Once we have the pipeline, various tests and evaluations can be done

DESIGN

Target scenario

- Goal: A representative scene where machine learning and graphics are both required
- We start with a small but typical case: human face rendering
  - Real-time face alignment/pose estimation using RGB video
  - Superresolution to improve performance/image quality
- Demonstration rather than product
  - Fancy graphics is not what we are after
  - We would like to show you the process to customize such a pipeline

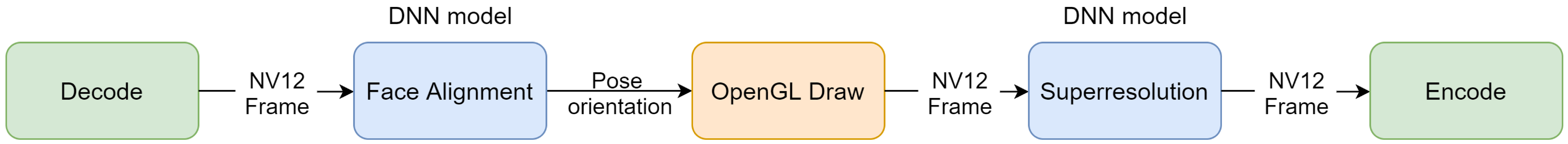
```
ffmpeg -hwaccel cuda -hwaccel_output_format cuda -i ../output/rio.mp4 -vf
scale_npp=1280:720,pose="img2pose.onnx":8,format_cuda=rgbpf32,tensorrt="./v57_720.onnx.trt8",format_cuda=nv12 -c:v h264_nvenc
-preset p7 rio_2k_out.mp4
```

DEMO

DESIGN

Pipeline

- We want to build a FFmpeg based full GPU pipeline
  - Data resides on GPU
- GPU encoding & decoding
- Two DL models

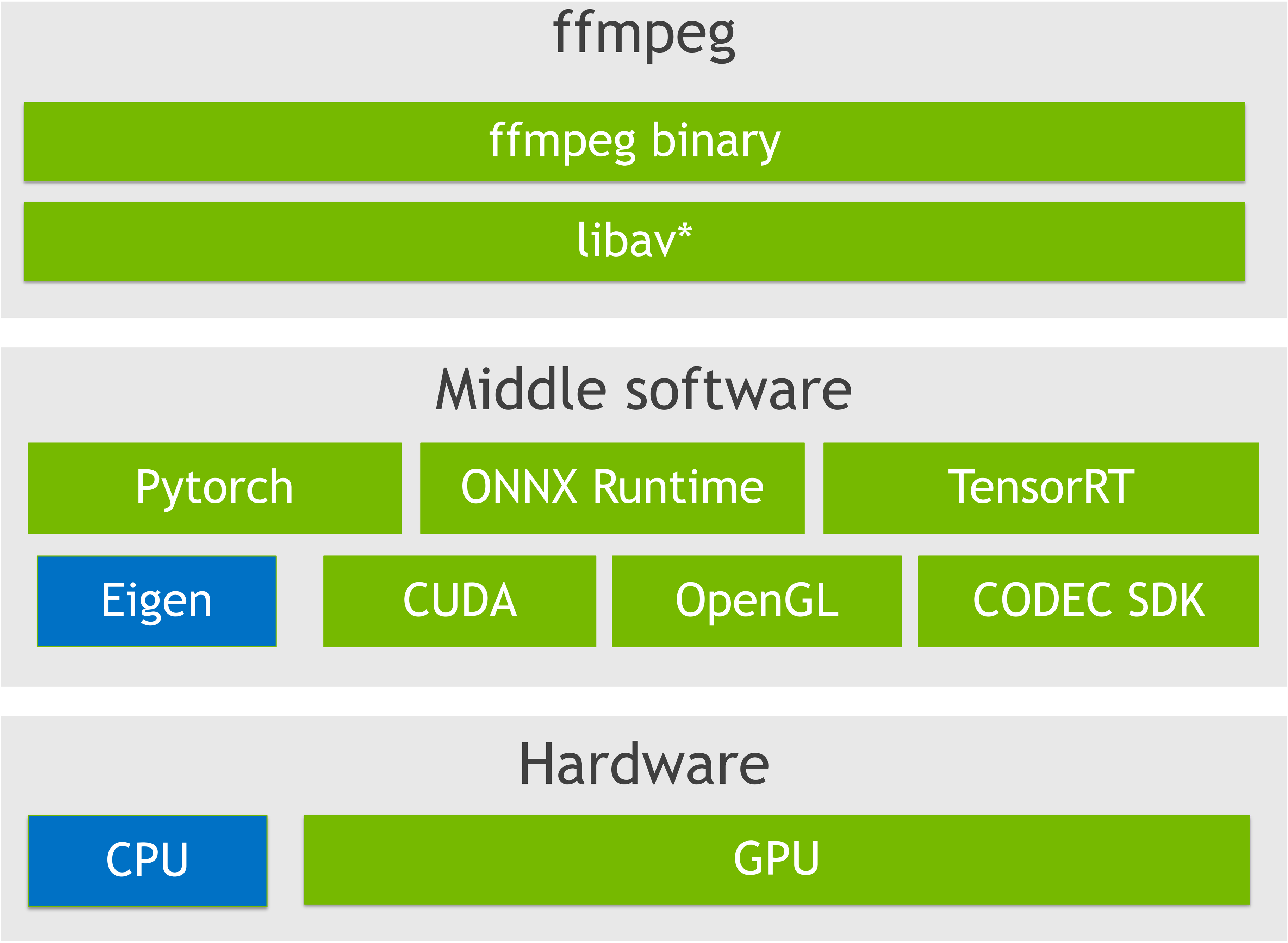


DESIGN
FFmpeg

- FFmpeg is the de-facto industry standard for video transcoding.
- If we want a unified live streaming platform, it must be built upon FFmpeg.
- FFmpeg processes video in a pipelined fashion.

Decode → Scale → Flip → Encode

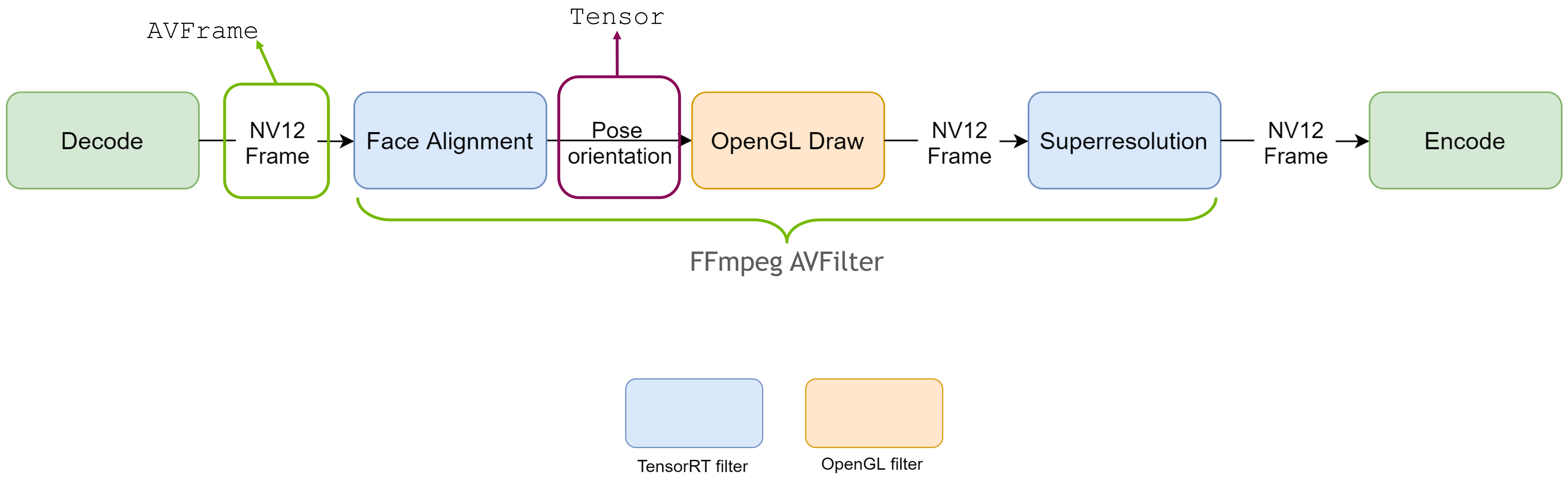
- FFmpeg uses filters to process decoded frames.
- Full GPU pipeline -> implement GPU HW filters
  - An OpenGL filter for rendering
  - A TensorRT filter for inference



DESIGN

Pipeline in FFmpeg

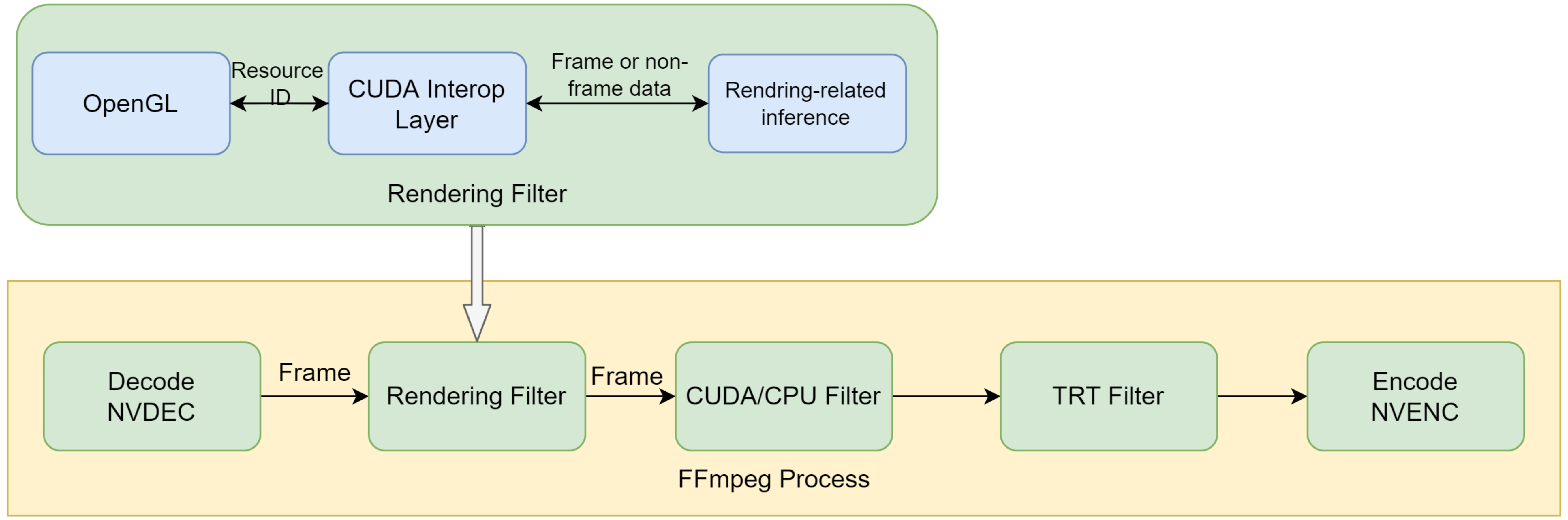
- Problem: ffmpeg is only designed to flow frame data between filters
- The output of face alignment are tensors, which is hard to fit in `AVFrame`
- Enable non-frame data flow requires fundamental low-level modification on FFmpeg



DESIGN

"Meta-rendering filter"

- To avoid non-frame data, we decided to put face alignment & OpenGL into one filter
- "Meta-rendering": inference before rendering
- Both inference and rendering need to access the GPU frame data

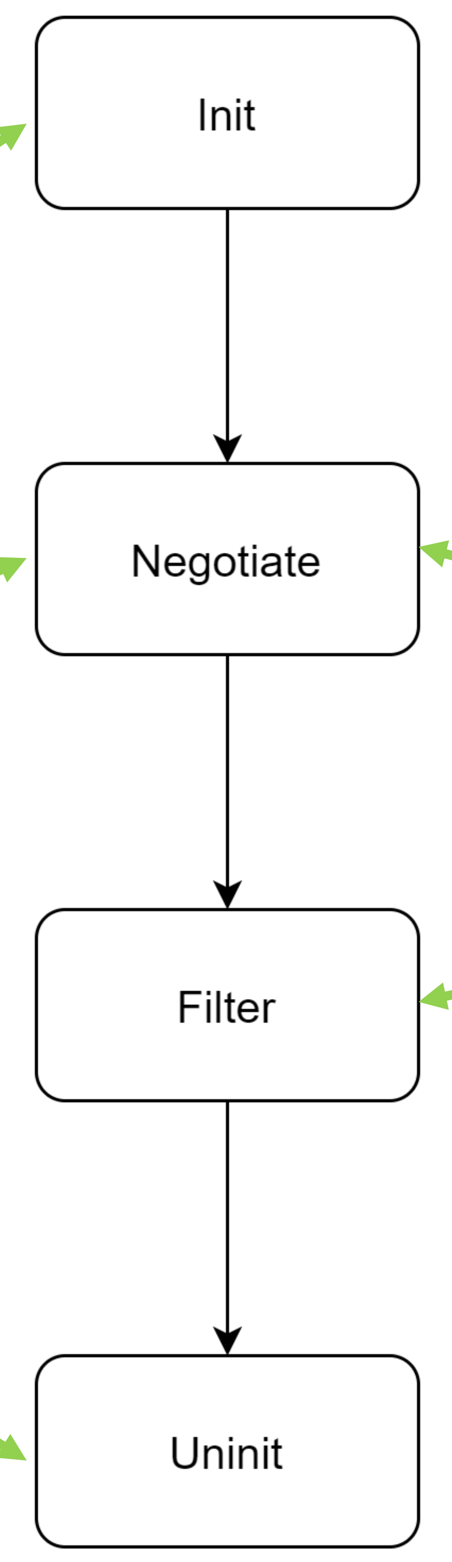


FFMPEG

Custom GPU filter

- Define an AVFilter variable
- Implement the necessary function pointers
- Refer to other GPU filters, e.g. scale_npp, thumbnail_cuda

```c
typedef struct AVFilter {
    const AVFilterPad *inputs;
    const AVFilterPad *outputs;
    const AVClass *priv_class;
    int flags;
    int (*init)(AVFilterContext *ctx);
    void (*uninit)(AVFilterContext *ctx);
    int (*query_formats)(AVFilterContext *);
    int priv_size;
    int (*activate)(AVFilterContext *ctx);
} AVFilter;
```

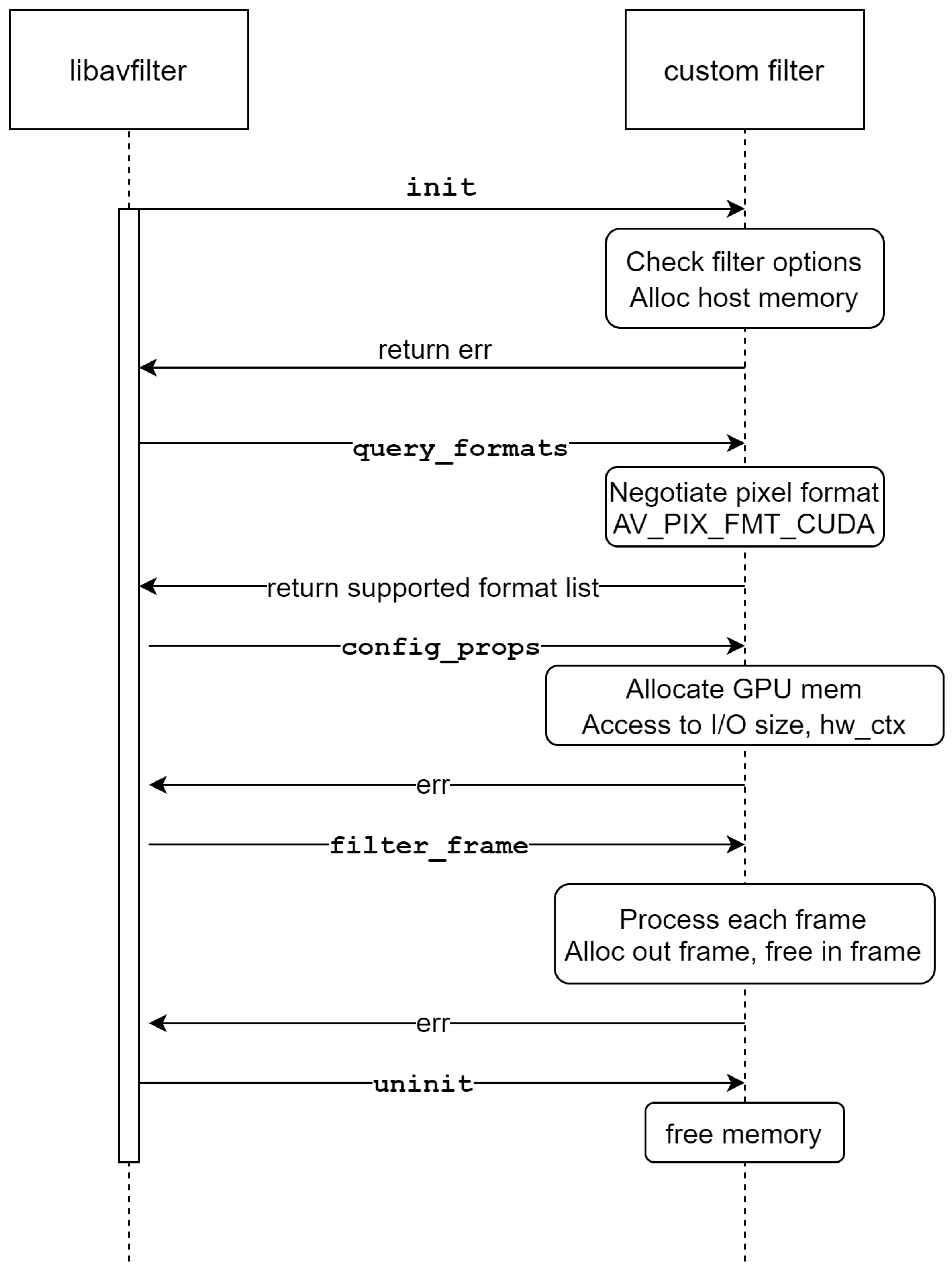



```c
struct AVFilterPad {
    const char *name;
    enum AVMediaType type;
    int (*config_props)(AVFilterLink *link);

    int (*filter_frame)(AVFilterLink *link, AVFrame *frame);
    ...
};
```

FFMPEG

Custom GPU filter



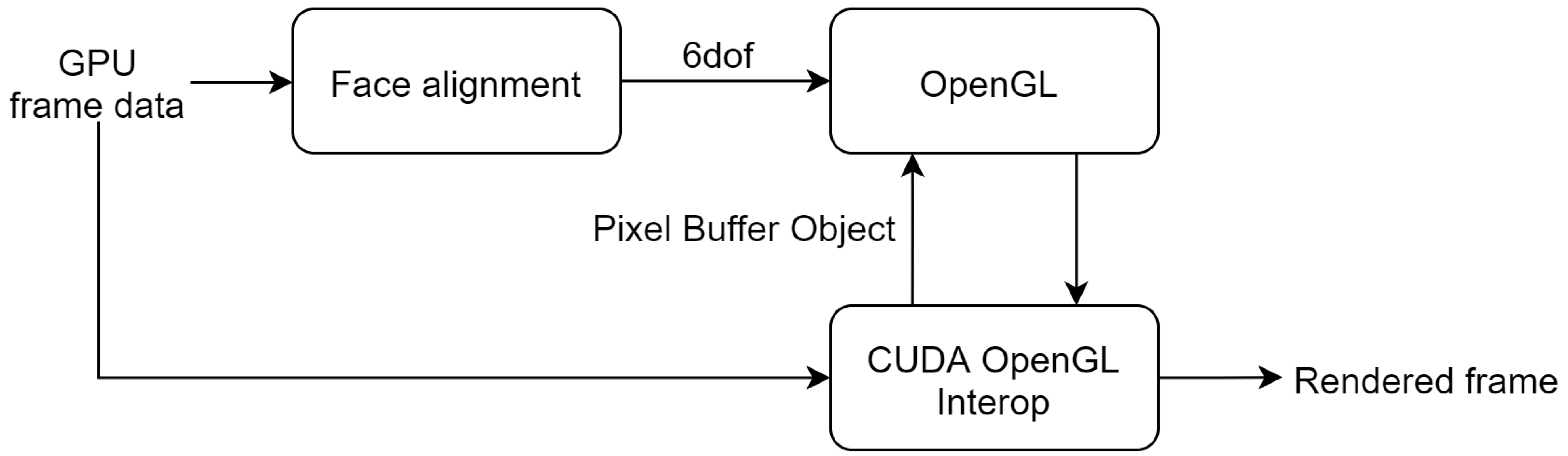
FFMPEG

CUDA Context

- In FFmpeg, by default, CUDA context is created by libavutil using driver API, and passed to your filter
- Most libs like TensorRT uses runtime API
- We need to make sure that CUDA runtime uses the CUDA context created by ffmpeg
- Push CUDA context right before any CUDA calls, and pop it immediately after
- Make sure CUDA context is popped before OpenGL calls
- Check if CUDA context changed when running into `invalid resource handle/invalid memory access/cudnn status mapping error`

RENDERING FILTER

- We use a Faster R-CNN--based model (img2pose*) to regresses 6DoF pose for all faces in the photo
  - 6DoF = $\{v_{rot}, v_{tran}\}, |v| = 3$
- The pose will be used as the orientation to draw the mask in OpenGL



- *img2pose: Face Alignment and Detection via 6DoF, Face Pose Estimation

POST PROC

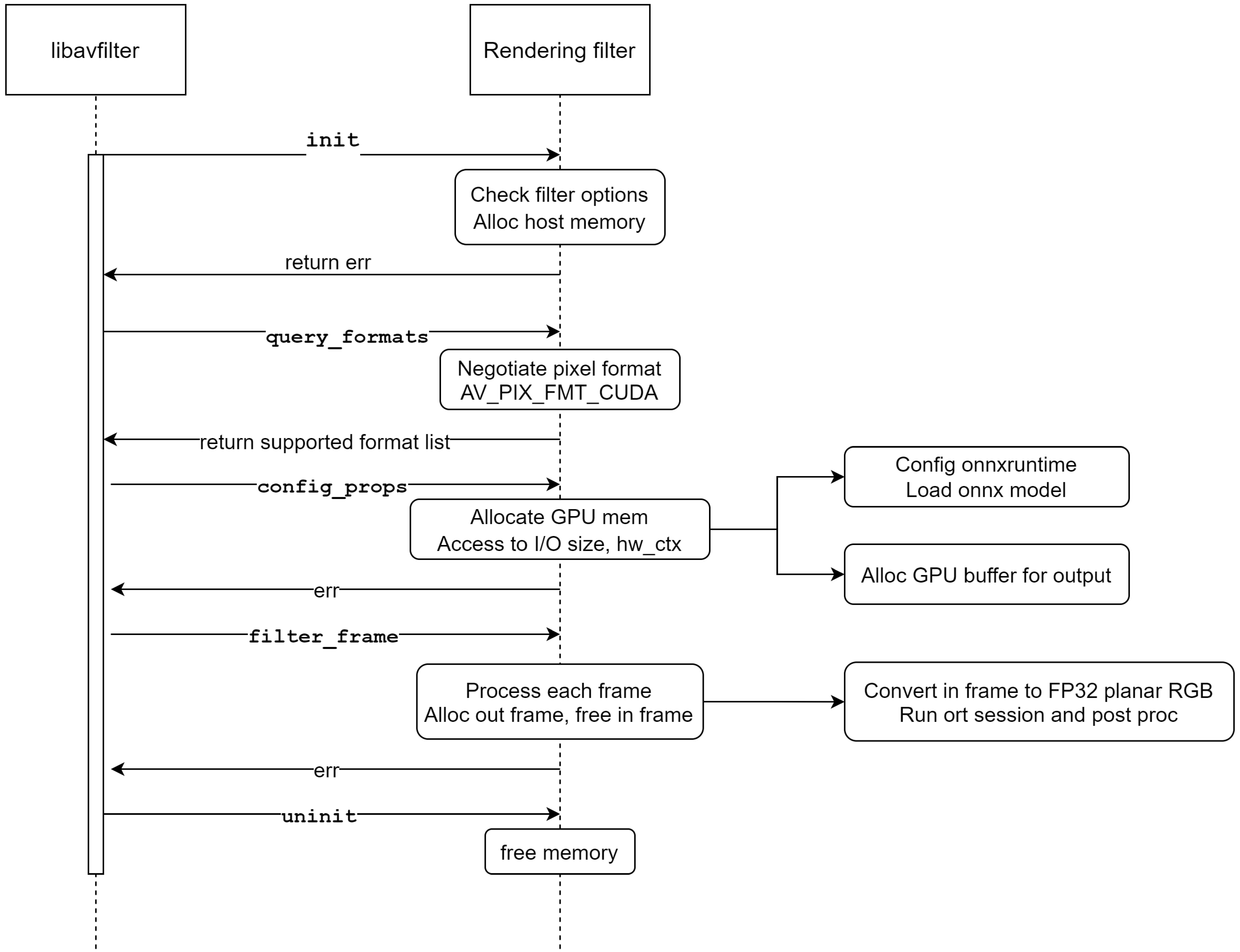
- Accelerating the neural network on GPU is a common task, but post proc needs special treatments
- Post proc in img2pose is mostly on CPU (Numpy)
- Except nms, the conversion & projection are operations on small matrices (e.g. [68, 6])
- Numpy's perf is acceptable in this case
- After evaluation, we used a hybrid method:
  - CPU: Eigen to implement numpy operations
  - GPU: NMS from libtorch
  - Data is small, low memcpy overhead
- Almost real-time performance on TU102 (~25 fps)
- The acceleration is significant when there are more faces

image	Detected face number	Actual face number	Post processing time (w/o nms)	GPU nms	CPU nms (not stable)	Ort inference time
To4_jun	68	52	0.8 ms	2ms	35ms	38 ms
One_face	3	1	0.07 ms	2ms	16ms	38 ms

image	PyTorch time	Accelerated
To4_jun	375 ms	40 ms
One_face	43 ms	40 ms

RENDERING FILTER

Add inference to the filter

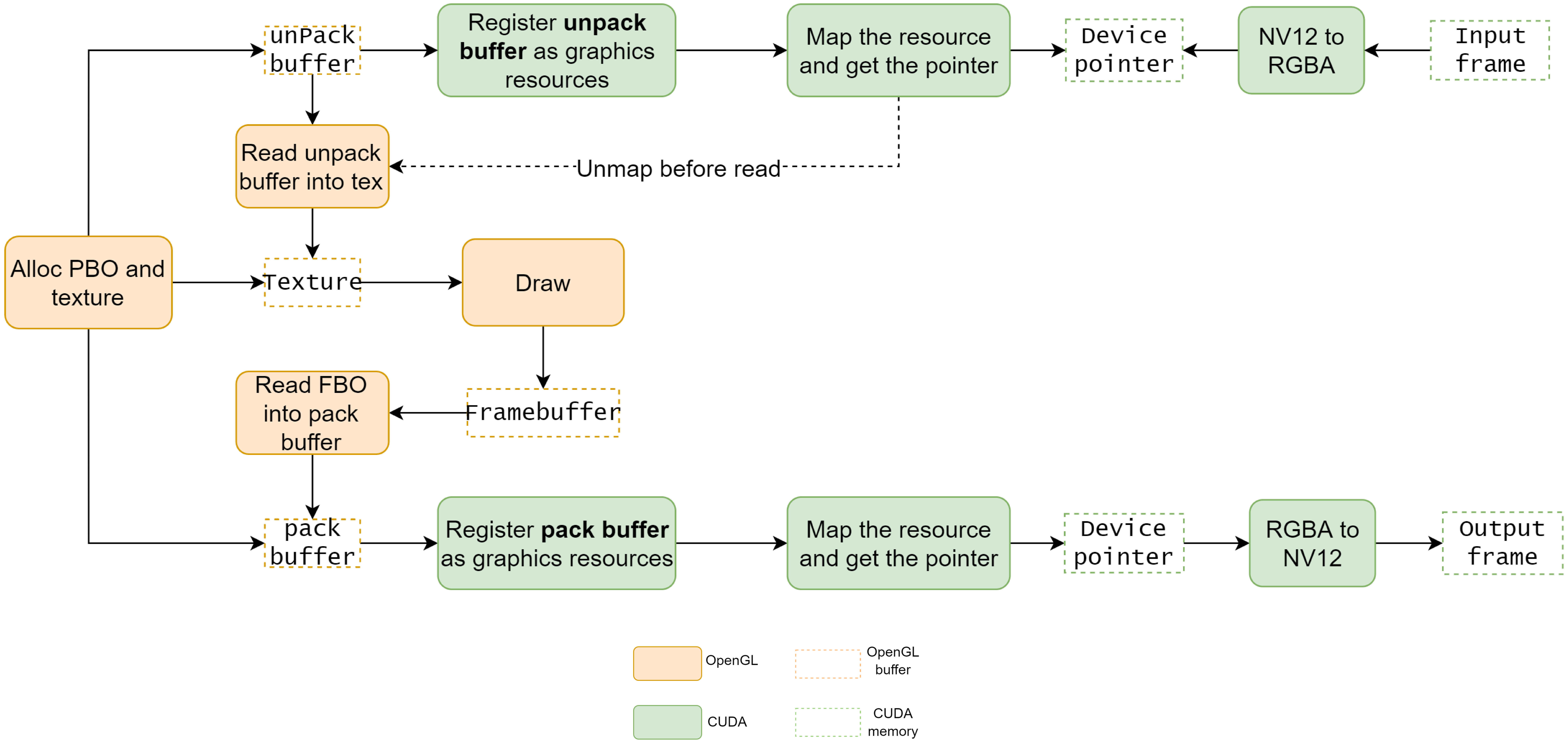


RENDERING FILTER

CUDA OpenGL interoperability

- OpenGL needs to draw on the GPU frames
  - CUDA memory ⟷ OpenGL buffers
  - Bypass CPU
- CUDA uses familiar C memory management
- OpenGL is state-based and stores data in abstract generic buffers called *buffer objects*
- CUDA/OpenGL interop uses one simple concept:
  - Map/Unmap an OpenGL buffer into CUDA's address space
- The interop uses Pixel Buffer Object (PBO) as relay
  - `GL_PIXEL_UNPACK_BUFFER/GL_PIXEL_PACK_BUFFER`

OPENGL INTEROP



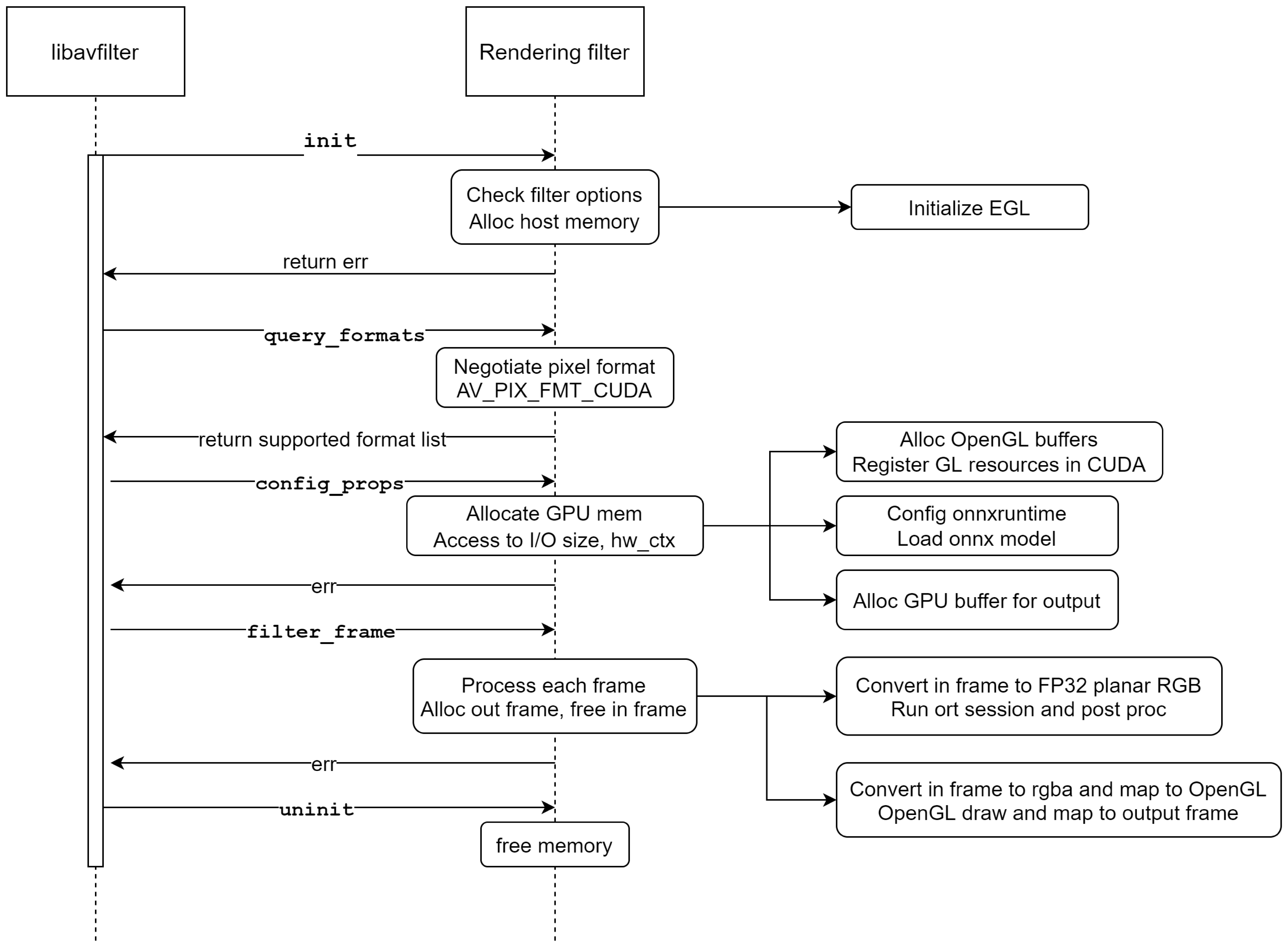
OPENGL INTEROP

Procedure

- Allocate PBO and texture with the size of the frame
- Map the PBO to CUDA memory
- Write the frame from CUDA to the mapped memory
- Unmap the PBO and create texture from PBO
- Draw the content in framebuffer
- Read the framebuffer into another PBO (pack buffer) and map the PBO to CUDA memory
- Write the memory to output frame

FFMPEG

Meta-rendering filter



RENDERING FILTER

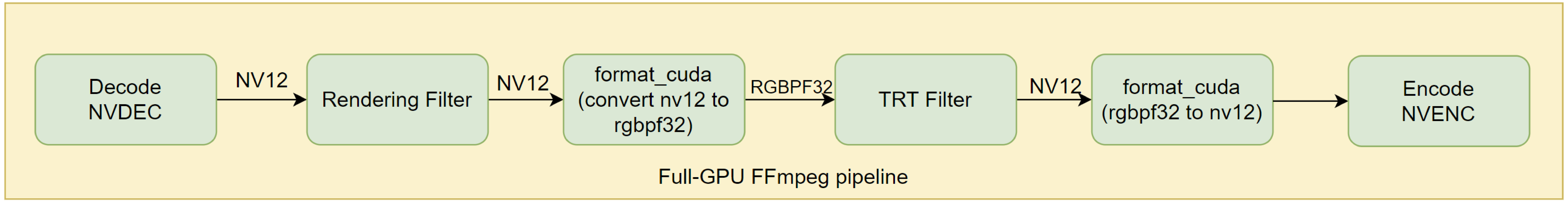
Asynchronicity

- Asynchronocity is important to GPU performance
- What's asynchronous:
  - CUDA kernel launch and execution
  - TRT inference
  - OpenGL rendering commands
  - GPU encoding/decoding
- What's synchronous:
  - ORT inference
  - Map/Unmap graphics resources in CUDA

→ Rendering filter is essentially running synchronously

TENSORRT FILTER

- TensorRT only accepts NCHW (planar RGB), which is not implemented by ffmpeg
- We add a new pix_fmt rgbpf32 (RGB planar float32)
- A format_cuda filter is added to convert frames between nv12 and rgbpf32
- `ffmpeg -hwaccel cuda -hwaccel_output_format cuda -i <input file> -vf scale_npp=1280:720,format_cuda=rgbpf32le,tensorrt=<model path>,format_cuda=nv12 -c:v h264_nvenc <output file>`



PERFORMANCE

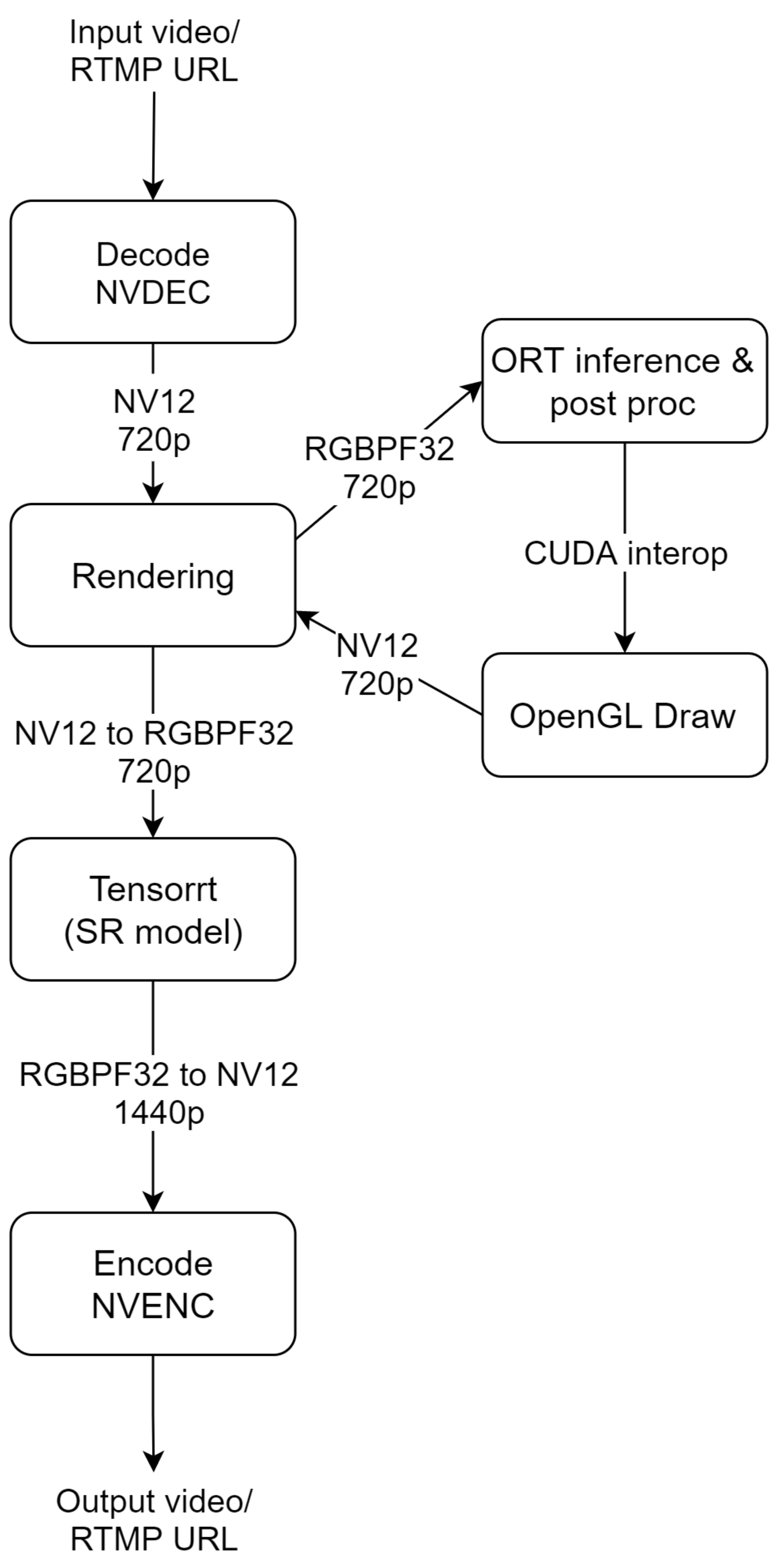
- Input video: 720p30 with multiple human faces
- SR model from NVIDIA Maxine, running in FP16
- Img2pose run by onnxruntime in FP32

Device	A10	T4	A2
FPS (w/o SR)	31	12	9.4
FPS (with SR)	25	10	7.3
2 sessions (w/o SR)	29	11.5	8.4
2 sessions (with SR)	25	9.5	6.9

Pipeline performance

Device	A10	T4	A2
Img2pose network (fps)	32.7	12.6	9.5
Post-proc (fps)	> 5000		

Model inference performance



OUTLOOK

- We wish the project can become a platform for all-in-one cloud-rendering
- Once we set up the pipeline, various data can be tested:
  - Throughput in different resolutions/devices
  - Output video quality in different bitrates
  - Costs/scaling projection
  - Performance analysis & improvement
  - Comparison with other solutions
- We are still developing new cases and features
- The project is now opensource, stars are welcome!
  - https://github.com/NVIDIA/FFmpeg-GPU-Demo


NVIDIA