trdrop

T(ea)rdrop - a video analysis software

Alexander Isenko
Mujo Alic

1 Abstract

Offline feature extraction from streams in a single pass with automatic parallelization is a useful structure for several applications. We showcase our interpretation of this concept on a video analysis software by creating a type based parallelization with a transformation pipeline and external coupling. This allows a race-conditions-free, comprehensible interface to add features without depending on previous tasks while maintaining a sound performance.

2 Introduction

trdrop - pronounced [*'teə(r),drap*].

The gamer of today wants to buy the best hardware to get his money's worth. One of the console benchmarks is to run games at least as smooth as their competitors. To test this without bias, one needs to capture the raw video stream and perform several calculations on the uncompressed video to detect framedrops and a tears. We present a configurable command line tool to analyze multiple big, raw videos in parallel, insert a visual representation for the outcoming data and encode the videos in a single pass.

3 Features

The following features are included in **v0.1**:

- determine the real fps of the incoming video files with a sliding window of the capture rate
- show the fps as text in the video
- export the resulting video into a youtube friendly format (google-terms)
- create a csv-log with the framerate and every tear + fps
- visualize the fps in a plot
- import up to 2 `.raw` video files with a size greater than 15 `GB`
- analyze the videos in parallel

4 Program Diagram

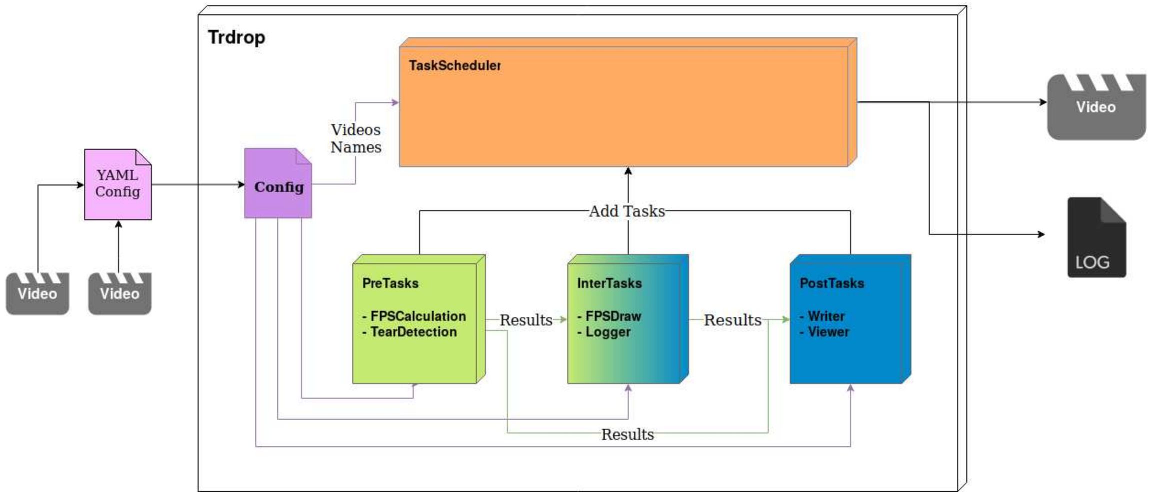

Figure 1: Main system diagram

The program has three main concepts:

Config sanitizing - rest of the programs deals with data in a valid state

Tasks - externally coupled separation of *model* and *view* for every algorithm allows for a parallelizable computation

TaskScheduler - configurable pipeline which traverses the videofiles in a single pass and applies tasks in parallel to it

4.1 Config

The program is configurable through a *YAML* file as seen in Fig.1, where you can customize the fps calculation, like the fps-refresh rate, fps precision and other options. The *Config* parses this file, checks the inputs for sanity and forwards it to the internal logic. The forwarding is implemented with **control coupling**.

```
using pretask = std::function<void(const cv::Mat & prev
                                 , const cv::Mat & cur
                                 , const size_t currentFrame
                                 , const size_t videoIndex)>;

using intertask = std::function<void(cv::Mat & res
                                   , const size_t currentFrame
                                   , const size_t videoIndex)>;

using posttask = std::function<void(cv::Mat & merged
                                  , const size_t currentFrame)>;
```

Figure 2: Interface definition of the three tasks

4.2 Tasks

This proposed division of an algorithm into tasks in Fig.2 demands a logical separation for the calculation and modification of the resulting video. It allows for an automatic parallelization of the calculations because of the external coupling down the pipeline. The tasks are divided in three kinds, which support the source-sink concept. They are added to the *TaskScheduler* which runs them with the respective arguments in the order determined by their type.

4.2.1 PreTask

The *pretask* described takes two videoframes to use for calculations, the current frame index and video index which activates the respective configuration of the task. This task has no access to modify the incoming stream of data, but may use external coupling to transmit calculation results down the pipeline. This fullfills the concept of a **source**.

4.2.2 InterTask

The *intertask* defines the intermediate task, which has the access to modify the videoframe for each video seperately with their respective configuration. It may use information from *pretasks* and modify the incoming datastream, but can also forward information to *posttasks*. This fullfills the requirements of a **source** and **sink**.

4.2.3 PostTask

The *posttask* gets access the merged datastream and can modify the resulting videoframe before it gets written to the disk. It may use all information passed down to it, but no previous task can be dependent on it's calculation. Therefore this is task is defined as a **sink**.

4.3 TaskScheduler

After the creation and coupling of the tasks, the *TaskScheduler* implements the single pass pipeline in Fig.3.

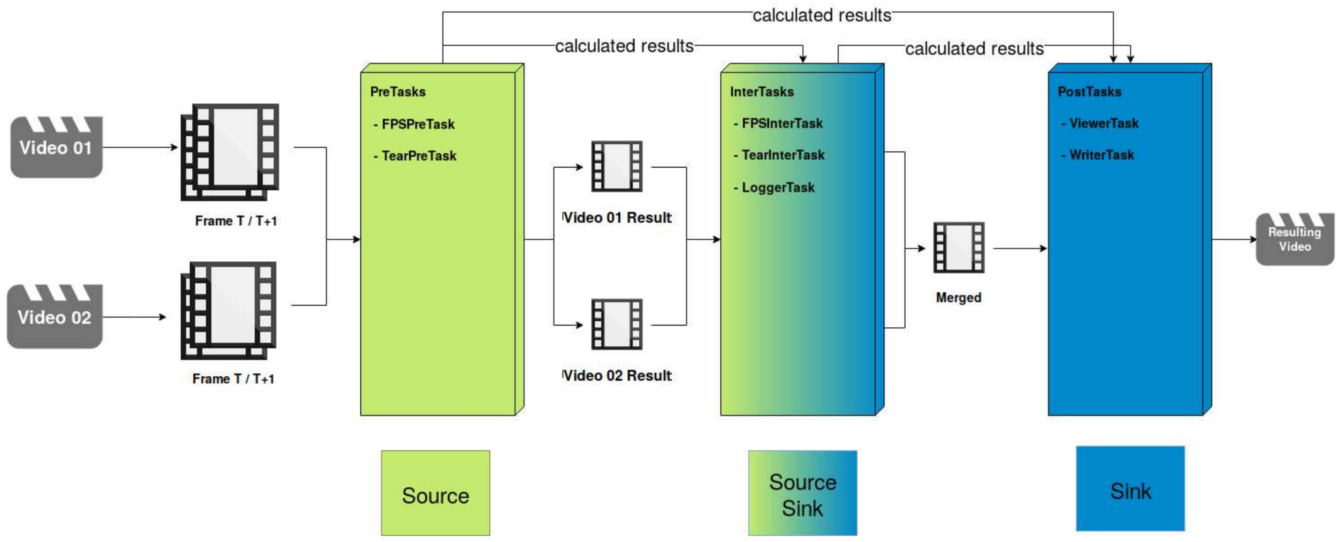

Figure 3: Task Pipeline

It has always "videos times 2" frames in memory and forwards them to the *pretasks* in parallel. The frame to modify is always at time `T`, because it will not be necessary for the next round of calculations.

These will be passed to the *intertasks* and modified accordingly. A single asynch call will be made for every video, but not for every *intertask*, because these tasks are not commutative.

The modified videoframes will be merged by the *TaskScheduler* into one frame and transferred to the *posttasks* which will be applied sequentially.

After this round the frame at time `T + 1` will be the new frame `T'` and the next one will be loaded as `T' + 1`. This loop aborts if any of the videos don't have any frames left.

The intetion behind this scheduler was to abstract the looping through multiple videos and provide an interface where tasks can be defined without explicit iteration through frames.

5 Why not do it another way?

In this paragraph we're going to stand up to some design decision and shortly explain why we took them.

5.1 Callbacks vs. explicit Either data object for task communication

The idea behind the `Either` object were tasks that could fail, or not trigger because of certain events (e.g sleep for n frames). Therefore a structure where we can call `successful()` was ideally suited. The choice for `Either` instead of `Maybe` was to be able to return an error value (e.g "still calculating fps").

```
if (fpsPreTask.result.successful()) {
    framerates = fpsPreT.result.getSuccess();
}
```

Figure 4: Accessing the fpsPreTask calculation

Next to the explicit nature and descriptivity of this branch (Fig.4) in the main loop, we made the decision to make the *tasks* responsible to store their results, so it would be possible to create a task without dependencies to the outer scope. Anybody who wants to get the result, can ask the task by themselves.

In fact, we have an exceptional callback solution for the *LoggerTask* (Fig.5), but this happend because we wanted to define to `string` representation of the logged values explicitly. These lambdas are not bound to the sources of the tasks and instead use the current value of the variable that stores the result. This way we don't have to check every time if the *fpsPreTask* is successful, because the code in Fig.4 updates it.

```
std::vector<function<std::string()>> convertions;
convertions.push_back([&]() {
  return string_format("%5i", scheduler.getCurrentFrameIndex());
});
convertions.push_back([&]() {
  return string_format("%6." + std::to_string(config.fpsPrecision) + "f", framerates);
});
LoggerTask<CSVFile> loggerT(config.logFile, 1, file, convertions);
```

Figure 5: Creating the LoggerTask (shortened namespaces)

5.2 Automatic parallelization with DAGs

Directed Acyclic Graphs are a very handy abstraction for complex dependency solving problems, but in our case we have the luxury to know that our computation is always linear, e.g we start with the first frame and end with last one. We also know that any computation has to run *atmost* every timestep with an input of two frames (this may even be desired, even if the dependent result does not change).

This domain specific knowledge allowed us to create the separation of three different tasks kinds which have different levels of parallelsim. If every computational result is initialized with a sane default value, we can define tasks that don't have any race conditions or wait times for unfinished calculations.

It may be true that with DAGs we could be a little bit faster, because some *intertasks* of `video01` may start before the *pretasks* of `video02` are finished, but the real bottleneck is the encoding of the video in the desired codec. Instead of this little speedbump we have direct control over the way how the task behave and how often they are called.

5.3 Do we really need three different tasks?

The decision to use an interface for three tasks came gradually as we expected to work with only a *pretask* and *posttask* which already provides most of the advantages of in Chap.4.2. The modification of the multiple videos in a non-sequential manner urged us to introduce the *intertask*, which allows partial parallelization between different videos.

Because of the externally coupled interface we provided, we expect to cover most of the future task and may introduce the **source** concept to the *posttask* as well as the **sink** concept to the *pretask*, if it seems fit.