COMP 50CP Project: Parallel Video Content Matching

Matthew Yaspan, Isabelle Sennett, Mitchell Katz, Alex King
December 11, 2015
Tufts University

Design

We have implemented a *video content matching* program; effectively a search-by-video reverse search engine. This is an interesting and open problem in computing, and it has very practical applications in areas such as copyright protection. Our program is written in Python, using the MoviePy library to use FFMPEG for video frame decoding. We compare frames directly as NumPy RGB arrays. We use the multiprocessing module for better parallel processing speed. We use the Flask framework to create a web-based user interface that interacts with our comparison engine.

To match the content of video, there are several possible approaches. Many videos include video and audio, so some sort of programmatic matching of video frames and/or audio needs to take place. We chose to focus on video content matching rather than audio content matching. In order to detect similarity between two videos, the content needs to be mathematically represented, and two representations need to be evaluated for closeness.

We chose a straightforward approach for this: match videos by matching a sequence of frames. To compare one video frame to another, we chose a rudimentary algorithm that was easy to implement, easy to test, and somewhat easy to optimize. We use the root-mean-square error method for finding the average pixel difference across two RGB buffers of the same size and width. Identical images will receive a score of 0.0, indicating an average difference of 0 across all pixels. Completely dissimilar images (a black image and a white image) will receive a score of 255.0, indicating the maximum difference possible across R, G, and B for all pixels in the image.

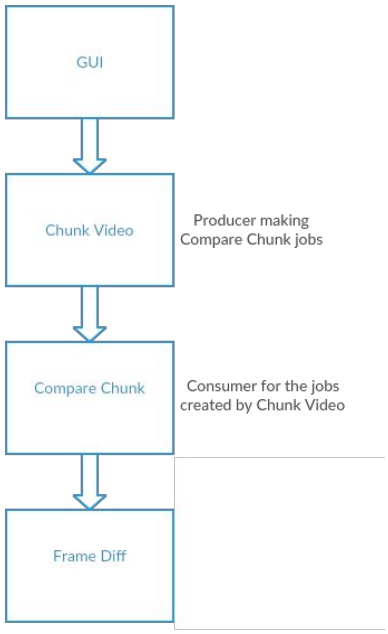
Our project defines its video database as a local directory of MP4 video files. After the local server the client can go to the website and upload a video they want us to find matches of and pick a threshold value. The threshold value gives the user the freedom to decide how close of a match we are looking for and thus what to filter out. After the client passes an input video and threshold to our program, the *producer* module iterates over the database and creates several search jobs, each defined by a start and end frame number of a given database video. We then create a certain number of consumer processes to do the searching. Each consumer calls our *chunk_compare* module that searches for a query video in the specified database video.

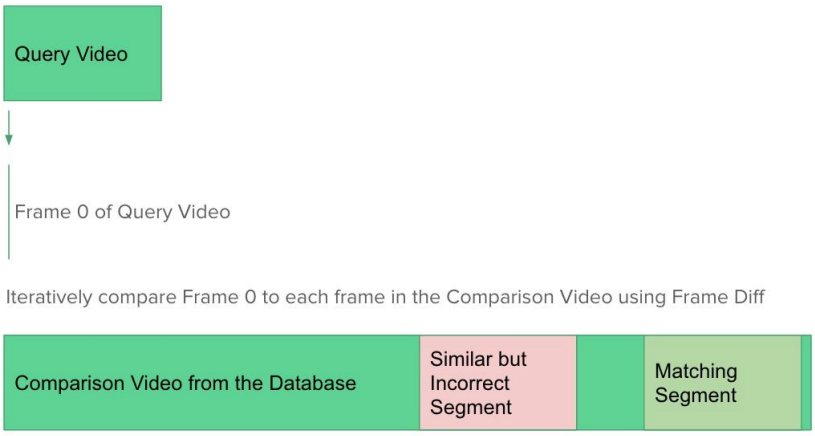
Within *chunk_compare*, the database video is scanned in search for a frame that is similar to the first frame of the input video. The similarity is defined by the threshold passed in by the client (typically in the 10-45 range). When a match is found, scanning over the database video temporarily pauses. The program now needs to determine if the entire query video exists within the database, so matching frames are searched in a loop lasting the entire length of the query

video. For example, if *chunk_compare* is searching for a 20-frame query in a 100-frame sequence and a match is found at database frame 30, the next 20 frames will be searched to see if the two windows of video content line up. This is measured by returning the mean frame pair similarity. Two identical sections of video will understandably have an average score of 0.0. We find that in practice, video noise or off-by-one errors can locate sequences with averages in the single digits that still correspond to a "true match".

Diagrams

Module Overview





Bugs, Snags and Other Design Reflections

The trickiest bug encountered dealt with sharing information across multiple processes, which is much harder than doing so with multiple threads. To calculate root-mean-square error between two video frames, we wanted to use a shared long integer to keep track of the total difference sum before averaging. However, when initializing a global value to 0 and forking multiple worker processes, each had its own value of 0, and therefore after they were joined, the calling process also still saw this value as 0. The solution was the *Value* object from multiprocessing. A *Value*, in this case declared as a C long integer type and protected by a multiprocessing Lock, will have a persistent value across multiple Python processes. This allows us to safely compute the difference between two frames while using 16+ unique processes.

We also had to carefully consider how we made search results available to the client while at the same time making sure we maintained a good level of abstraction. To accomplish this, we provide a web-site interface where the client can give us the data we need to start our search process and then they are redirected to a page that will give them the results. We wanted to make our search engine deliver results as they came in, rather than having the client have to wait for a return value of a static list of results after searching the entire database. The client code creates a multiprocessing queue and passes it to the the search engine backend. In return, the application gives the server the number of results that they can expect on the queue (we put a result on the queue for every search job executed; a search resulting in no match returns an empty result which the server code must filter out). It was important that we delivered the number of jobs back to the server, because otherwise it would be difficult to design a solution to signal to the server that the search of the database was complete and no more results could be expected. It could have also been done with a sentinel value being placed on the results queue when all other jobs completed, but checking for that condition would have required another global counter, and/or another forked process. We felt that returning the number of results was more elegant than forcing the server to check for a certain sentinel value.

Ultimately, the queue solution did work quite nicely to allow the server to do whatever it wanted to with results. If it wanted to simply pop things off of the queue and append them to a list for interpretation elsewhere, it could do that. If it wanted to print them live, it could do that too.

Overall, the design that we created for this assignment was very strong, and we did not have to make any significant architectural modifications after we began programming. The only area of our design that was not well established ahead of time was the explicit communication between the program and the user interface, and exactly how the website would display results to the client. Before we could get a better sense of that we had to not only figure out what was feasible but also learn GUI programing which none of us had experience with prior to this project. However, we were able to create a flexible “live” solution that very much improves the usability of our program.

Analysis and Outcome

Several goals of our project were satisfied:

- We *can* search a database of video files and recognize a match of content, with a variable definition of how close a match must be

- We *can* return results back to the client as they come in by livestreaming them on the web-page

- We *can* speed up the search process with predictable speed improvements for every additional CPU core we have at our disposal

- We *can* search for content in a database in an amount of time similar to the length of the video database (i.e., it is not an order of magnitude slower)

The last point is especially impressive considering that all of our code is implemented in Python, without any manual Cython-like optimizations. Additionally, we tested our code almost exclusively on consumer laptops with two CPU cores.

There are some qualifications to the above claims, though:

- Though we can recognize a match of content, our frame comparison method that allows us to do this is naïve, and does not hold up well to cropped, letterboxed, or otherwise transformed video content

- The total database search speed is inversely proportional to the query video's length; if a very short query is made into a very long database, the search will take longer

- While parallel searching demonstrated a performance increase, and parallel frame comparison did as well, we found that we could better optimize frame comparisons into small single-process jobs rather than multiple-process ones

We ended up achieving a product that is only marginally more developed than our predicted minimum viable product. There were not any monumental bugs that prevented us from making progress, rather, sometimes we chose not to meet the deadlines of our timeline. Fortunately, our design is well-modularized and would be very easy to extend. For example, another frame comparison module could be written and swapped in, with no other changes being necessary.

Division of Labor

Each group member worked together to develop a clear and modular design strategy. Alex was the principal contributor to design documents with contributions from all other members, Isabelle worked on the GUI module, Matthew wrote some framework code for the chunk_video. Ultimately, that module became our *producer* module which was written by Mitchell and Alex,

Mitchell worked on the *chunk_compare* module, and Alex worked on the *framediff* module. Each group member helped create and organize the presentation and demo.

By using a private GitHub repository, we were able to each work on different modules. Because of the clear contracts defined by each module, each was able to be completed without having to make modifications to others.

Organization

Files and subdirectories in the src directory:

- **framediff.py:** Frame difference module. Exports one function, *frame_rmse*, that returns the root-mean-square error between two NumPy RGB arrays.
- **chunk_compare.py:** Video chunk comparison module. Exports one function, *chunk_compare*, that returns a list of matching sequences between the queried video and the specified database video.
- **producer.py:** Parallel database search module. Exports one function, *server_entry*, that when given a multiprocessing Queue will return a number of results for the client to expect to be added to the results queue. In effect, this allows the server to call the program with a query video path, watch for live additions to the results queue, and know when no more jobs will be added on.
- **timeConvert.py**: Short module to convert between timestamps and second count
- **videoSearch.py**: Wrapper module to call producer from the command line
- **server.py**: Creates a server that maintains the web-based user interface. The server interacts with both the search engine and the client.
- **static:** the directory containing all CSS and Javascript code as well as the video database
    - **video:** a subdirectory in static containing the video database
    - **query:** a subdirectory in static where the video the user uploads is stored
    - **css:** a subdirectory in static containing all css code
    - **js:** a subdirectory in static containing all javascript code
- **templates:** the directory containing all HTML code templates for different pages
    - **home.html:** template for the home page where the user can upload a video and pick a threshold
    - **about.html:** template for the about page which has a variation of our design story
    - **help.html:** template for the help page which outlines how to use videosearch
    - **streaming_results.html:** template for the results page
    - **base.html:** a base template that contains element that will be on every page

TestFiles: includes some of our files from intermediary stages in our project

How To Run

Video Search relies on the following Libraries (which can be installed via PIP):

- MoviePy
- Pillow
- NumPy
- Flask
- SciPy

First make sure you have the Library's outlined above installed and that you are in the src directory of our project

There are two ways to run the application:

1. Through the GUI which is created as a web application using Flask
    - To do this run the command `python server.py` in console,
    - then open up a web browser and go to the URL `http://localhost:9090/`
2. You can also run the program via the console, using the command:

```
python videoSearch.py inputfile.mp4 path/to/database/ threshold
```

Threshold should be a percent value between 0-100.
With the directory system provided, this is a good sample test (run from src directory)

```
python videoSearch.py ../TestFiles/Sail7s.mp4 ../TestFiles/Sail/
30
```