Video Shortener

Grace Chang

> **This program takes a video and uses machine learning to shorten the video by cutting out “uncessessary” frames. The result is a shortened video that is not sped up yet still (supposedly) contains the main parts of the video.**

The inspiration for this program was feeling impatience for cooking videos and other types of video where I found myself wishing I could just quickly get to the final product.

Example Output:

These are different links to different shortened videos of the canonical Big Buck Bunny (https://peach.blender.org/) short video (`threshold` is how high of a score a frame should to make the cut to the new video).

As the threshold goes higher, fewer frames pass the test and the resulting video is shorters

Original (`9:56`): https://www.youtube.com/watch?v=NCsxpz6ro-I

Threshold 0.1 (`6:45`): https://www.youtube.com/watch?v=2ywc_QCUf3g

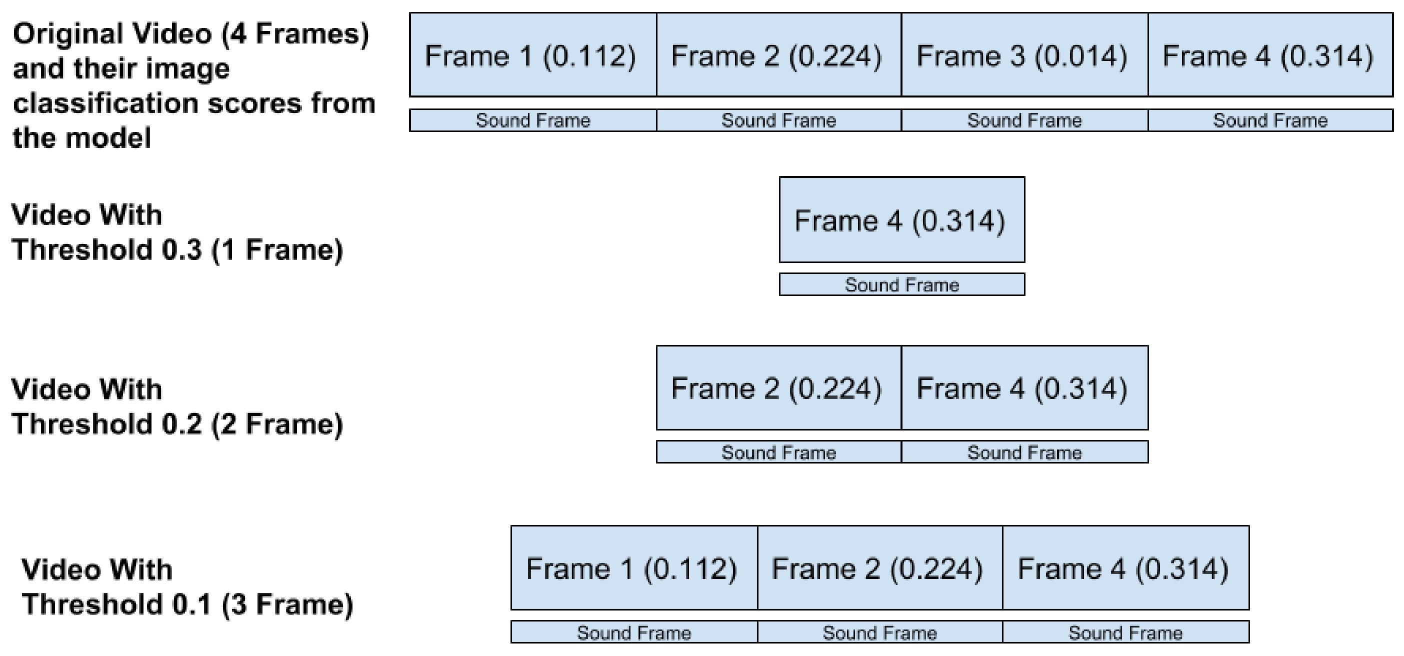
Threshold 0.2 (`2:40`): https://www.youtube.com/watch?v=CIZIj5WoalY

Threshold 0.3 (`1:01`): https://www.youtube.com/watch?v=JX8mUOmzCUo

Implementation

Split movie into frames, score each frame though a pre-trained machine learning image classification model. If score is higher than a certain threshold, save the frame and accompanying audio into a new video. Resulting video "should" be a video with only the "important" frames.

The higher the threshold, the higher the frame has to score in the model.



Notes/Disclaimers

The program uses `ffmpeg` for a lot of the video manipulations, and `ffmpeg` can be run in parallel.

Scoring happens through an API call from a personally hosted EC2 instance of **https://outcrawl.com/image-recognition-api-go-tensorflow**. The API can handle parallel requests.

Summary

Video Pre-processing

- Temporary directories are created to put the in progress files
- Sound is stripped from the movie and saved. This sound file will be later used to get the corresponding sound frame from the video.
- Movie is split into image frames from a specified hertz (using ffmpeg). A hertz of 1 will sample the movie at 1 frame per second, a hertz of 10 will sample the movie at 10 frames per second, etc.
- `ioutil.ReadDir` is called to get a list of all of the frame files

Pipeline

This happens for each frame file info from `ioutil.ReadDir` .

Stage 1) (`collectFileNames`) Get path name from the `io.FrameInfo` and output as stream

Stage 2) (`gatherPaths`) Convert each path name to a `score.MovieFrame` struct and output as stream

Fan Out

- Note: This is what the thread parameter affects.
- Create scoring workers for each thread specified (e.g. 2 threads results in 2 scoring workers)
- Scoring workers consume from the `score.MovieFrame` output from the pipeline and score each frame by uploading it to the scoring server
- Frames are scored independent from one another and is thus *data decomposition*
- While scoring workers are scoring, other parts of the program are filtering and moving files, resulting in *functional decomposition* along with the *data decomposition*.

Fan In

- All output channels from the scoring workers get fanned ino `score.FilterAndMoveAllOutput` which will collect all the channels into one and then run it through a threshold filter.
    - Threshold filters only passes (marks frame as good) the frames that have a score that exceed a certain threshold
    - If the frame's score exceeds the threshold, the file is moved to a separate directory and `ffmpeg` is run to gather that particular's frame's sound (a 10 hz video will create a second sound frame of 1/10 of a second of audio)

Pipeline

- Loop over channel of output of the fan in to collect the rest of the moved files

Video Saving

Audio is stitched together (using `ffmpeg`)

Libraries

`upload` : Handles all data uploads and API communication with the scoring server

`frame` : Handles image and sound processing that require `ffmpeg`

`score` : Handles post processing and other transformation from results from the scoring server.

Challenges

Non-Parallel Programming Challenges

- Running the tensor flow pre-trained machine learning model is hard to build from scratch because it relies on certain dependencies which can vary between operating systems. Thus, the remote API server was the most straightforward way to use the model.
- Multipart uploading for mime forms for making a `POST` call was on the complex side but needed to upload the video.
- There is no good video processing package in Go, so the recommended way is using `ffmpeg` , which is extremely robust. Drawback is the program ends up making bash calls in the code.

Parallel Programming Challenges

Some parts needed to be synchronous for the processing to be correct. Certain parts of the code, such as video preprocessing, needed to be run in a synchronous way. Operations on the video in its entirety such as saving frames from the video, stripping audio from the video, stitching audio, etc needed to be done to correctly process the image.

Try to fit function and data decomposition together to achieve more efficiency. Functional and Data Decomposition at the same time was needed because there were multiple longer steps. While some calls took longer than others, it would be optimal if (while the thread was waiting for an API response) we also were moving the file and collecting the sound from it.

Balancing ordered and not ordered practices matter. Some patterns (fan in/fan out) were fast and did not need to be ordered, but other parts (pipelines) preserved order. It was important to identify when the program needed to preserve order and when it didn't and when it was overkill to do a fan in/fan out for a relatively trivial task. Another balance was the need to keep things ordered, but the majority of the time things did not need to be in order. This was the case for frames, which could be processed out of order but needed to be ordered in the end. The way the program tried to accommodate this was to bake in the order into the filename, and sort the filenames in the end (which `ls` actually does for us for free).

Machine

- Laptop: MacBook Pro (13-inch, 2017) Processor: Intel Core i5 (2 cores)
- Processor Speed: 3.3 GHz
- Memory: 16 GB 2133 MHz LPDDR3
- Graphics: Intel Iris Plus Graphics 650 1536 MB
- Operating System: macOS Sierra 10.12.6
- L2 Cache Size per Core: 256KB
- L3 Cache Size: 4MB
- Number of CPU: 4

Results

Performance was tested on different lengths of videos (does the total number of frames affect overall speedup?), different threshold sizes (does the number of passing frames affect speedup?), and hertz (does the total number of frames affect speedup?).

Tested Scenarios

Videos

The following videos were used in the performance test:

- **Dog Dance Vine (0:04):** https://www.youtube.com/watch?v=Ss96Hff7OnE
- **Two Lynx in Ontario Have Intense Conversation (1:09):** https://www.youtube.com/watch?v=eaXmIPHrHmY
- **Polishing a Rusty Knife (3:54):** https://www.youtube.com/watch?v=3XW-XdDe6j0
- **Wreck It Ralph Trailer (4:42):** https://www.youtube.com/watch?v=KHQhp2cGZtE
- **Top 10 Cake Decoration Ideas (10:33):** https://www.youtube.com/watch?v=qNVfkAmAbu0

The Youtube videos were saved locally then run through the program. They are also available on my dropbox here: https://www.dropbox.com/sh/2dhcy49pda9w2es/AABej_94nDeZGNk7aA502n6ma?dl=0.

Thresholds

The following thresholds were used in the performance test:

- Pass frames with a score `> 0.1`
- Pass frames with a score `> 0.2`
- Pass frames with a score `> 0.3`

Hertz

The following frame hertz were used in the performance test:

- `10hz` (1 frames / second)
- `10hz` (10 frames / second)

Thread Count

The following threads were tested:

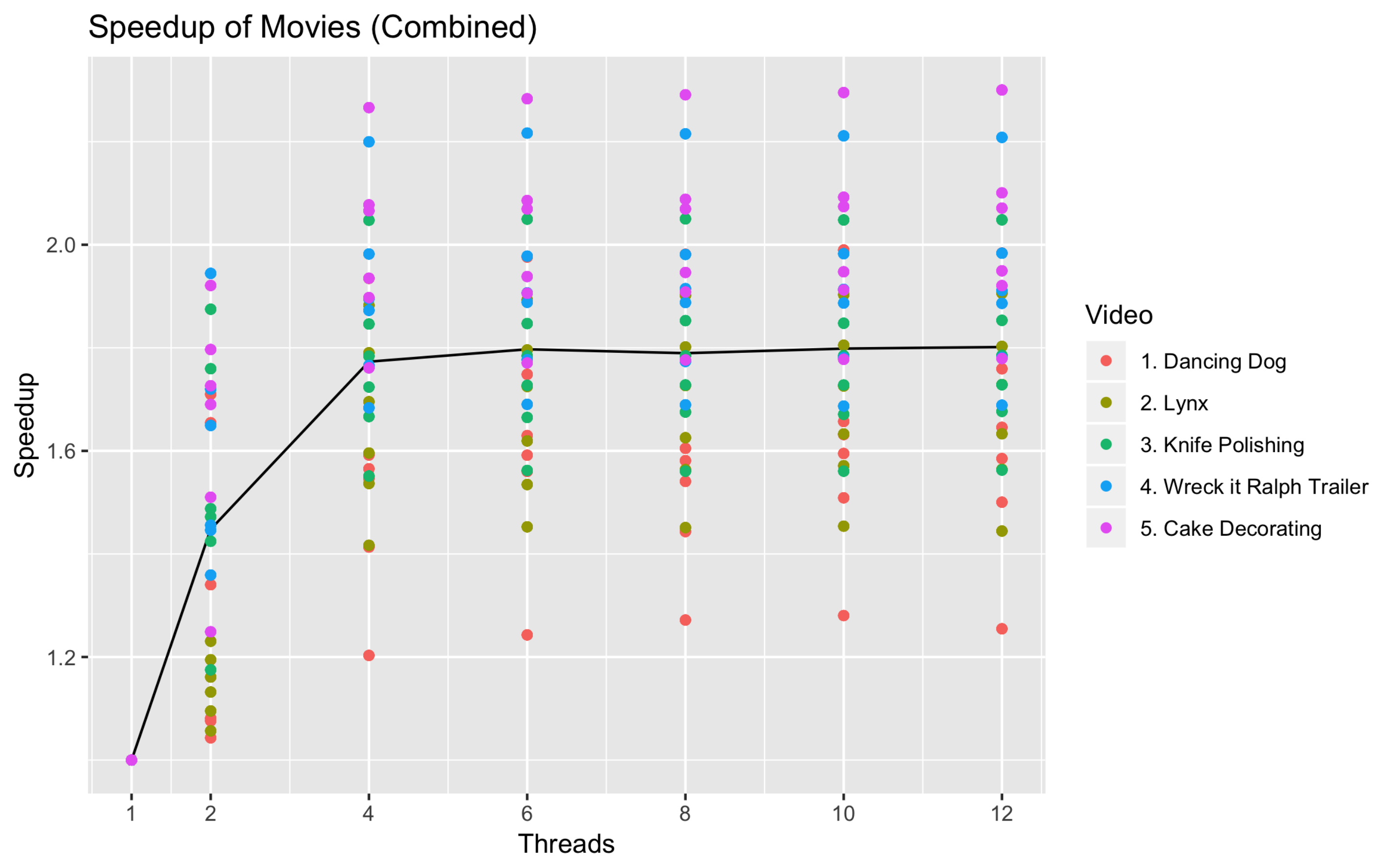
- 1 thread (sequential)
- 2 threads
- 4 threads
- 6 threads
- 8 threads
- 10 threads
- 12 threads

Scenarios

All combinations were tested, resulting in a comprehensive view of overall performance and speedup.

Results

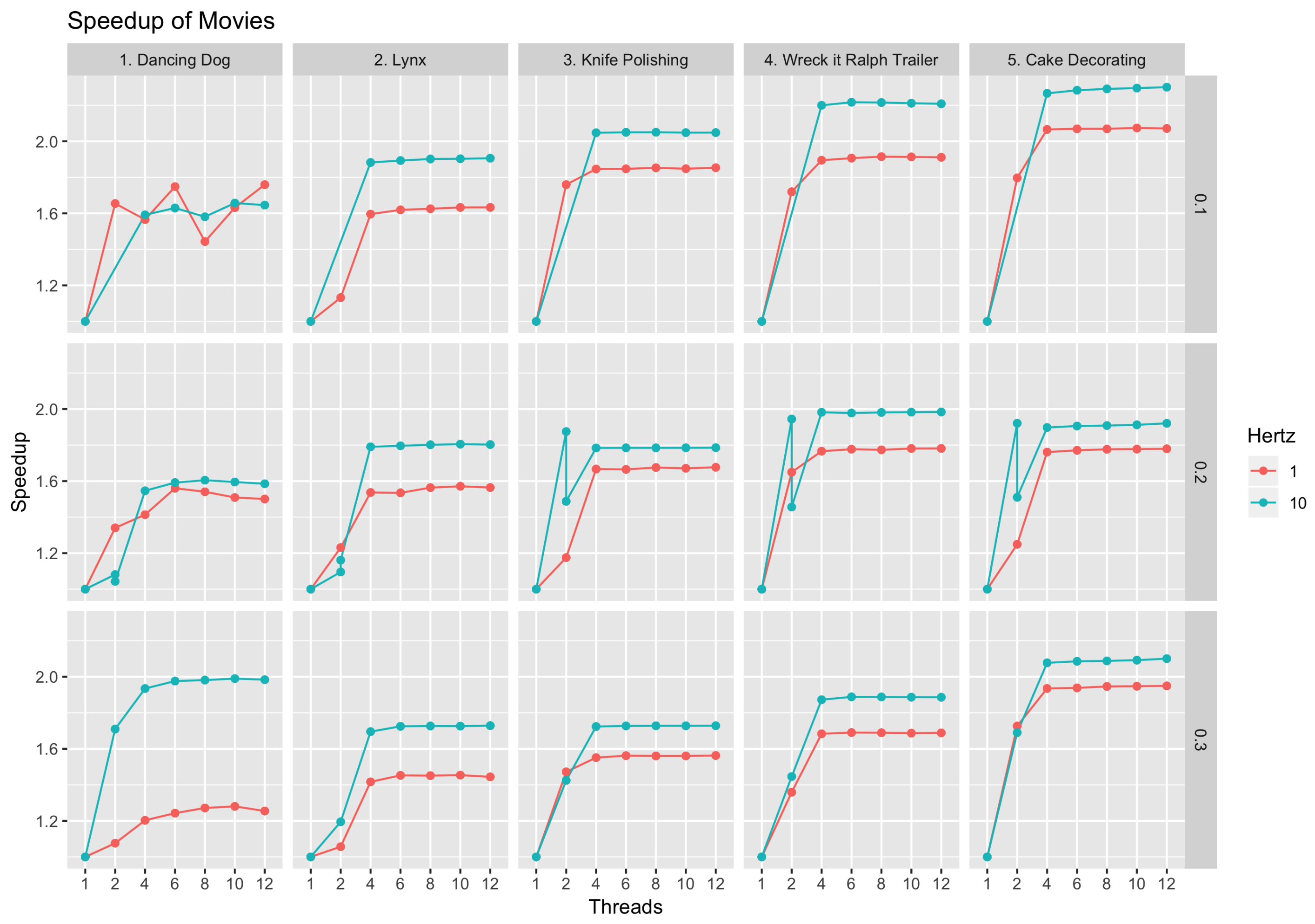
Combined Graph



This is a combined graph of all of the runs. The `1-5` of the videos are ordered from shortest to longest. We can see that there are general "bands" of speedup for each video, with the longer videos having more speedup than the shorter videos. Regardless, there is definitely a speedup from sequential to parallel.

We can see that performance improvement tapers off at around 4 threads. My computer has a 2 core, 4 thread processor, so it will be best optimized on 4 threads in the program. We can see this in the graph--after 4 threads, the performance tapers off because the number of threads is higher what the hardware physically supports. Of course, we can spawn more than 4 threads, but in the background the processor is just managing them over its 2 core, 4 thread processor.

Sectioned Graph



This is a plot of the same data as above, but separated into a grid. The y axis is the speedup from sequential and the x axis are the number of threads. Each row (band) corresponds to a different threshold score required to pass a frame. Each column corresponds to a different video. Lastly, each color represents the to hertz values tested.

This graph shows that as the amount of work increases, the speedup from sequential increases.

We can see from left to right that as the videos get longer, the speedup from sequential gets bigger. We can also see that more frames (10 hz) provides more speedup than fewer frames (1 hz). We also see that as we lower the threshold, resulting in more frames passing and more sounds frames getting creates, the speedup from sequential gets higher.

Hotspots and Bottlenecks

> **The graph from the previous section shows that as the shortening of the video becomes more computationally expensive, the speedup from sequential increases. This is a good sign because it shows that that the program is parallelizing the more expensive parts of the code.**

The main hotspots would be the most time consuming and repetitive actions that block other parts of the code from progressing/executing.

Notable Hotspots and Bottlenecks: (Some of the processing can vary from video to video so I have chose the knife polishing video as the representative video).

- Pulling the Sound from Video
- Converting Video to Frames
- Saving the Completed Video
- Scoring the Frames
- Moving the Frames and Cutting Sound Frame

Limitations

I was unable to parallelize some of the sound and video computations the require `ffmpeg` . Stripping sound took around `0.07` seconds on average for the knife video and I was not able to parallelize it. Converting the knife video to frames (1hz) took around `5` seconds and I was not able to parallelize that either. Lasting saving the video took around `0.03` seconds and that was not parallelizable either. I would consider these to be fixed costs that unfortunately need to be run in a sequential manner.

The program is also very susceptible to internet speed fluctuations, meaning that speed is often at the mercy of the internet speed and impossible to speed up.

Improvements

As I added more threads, the overall speed decreased for the frame scoring and filtering and sound clipping that I did parallelize. Scoring a frame consistently took around `120ms` with my current internet. Filtering a frame that passed the threshold (requiring sound clipping) took on average `65.3ms` . Filtering a frame that did not pass the threshold (and thus required no further action) took around `0.005ms` . Regardless of parallelism, these individual speeds were relatively consistent, and it was just a matter of strategic overlapping of commands.

1 Thread

- SCORE FRAMES `28987.39594 ms`
- FILTER AND MOVE FRAME AND POSSIBLY CLIP SOUND `4152.330532 ms`

2 Thread

- SCORE FRAMES (thread 1) `18502.405635 ms`
- SCORE FRAMES (thread 2) `18568.371077 ms`
- FILTER AND MOVE FRAME AND POSSIBLY CLIP SOUND (pipeline) `18568.448909 ms`

4 Thread

- SCORE FRAMES (thread 1) `17087.336887 ms`
- SCORE FRAMES (thread 2) `17225.461938 ms`
- SCORE FRAMES (thread 3) `17226.416195 ms`
- SCORE FRAMES (thread 4) `17294.788106 ms`
- FILTER AND MOVE FRAME AND POSSIBLY CLIP SOUND (pipeline) `17294.893994 ms`

There are a number of notable parts of this speedup. The first is that the scoring frames for the multi threaded runs is lower than the single threaded runs. This means that when the two threads are running scoring in parallel, it can overall reduce scoring by a little bit.

The other notable part is the jump in the filtering section. At first it might be confusing because it looks like the single threaded version is faster, but the reason the multithreaded numbers are a bit higher is because the timing is the duration of the pipeline stage and it starts filtering while the scoring is still happening. This means that this stage is waiting for the rest of the frame scores to come in. Thus, the ms for filtering is higher, but only because it is overlapping with the frame scoring. This is also another indication that we are succeeding in functional decomposition while the data decomposition of the scoring is happening on a different thread.

General Conclusion

Given the increasing returns of speedup as we added more computational complexity, and the timings that show that the ms were overall reduces when we run the actions in parallel, the main hotspots in the program were addresses in the parallel version.

Advanced Features

A number of advanced features were used to parallelize image processing, video processing, sound processing, and API calls.

Patterns

Pipeline

- `main.go` line `150` is a pipeline of `collectFileNames` -> `gatherPaths` . (Each spawn their own goroutine)
- `main.go` line `157-169` is a pipeline of `fanInOutput` -> `scorer.FilterAndMoveAllOutput` as both spawn their own goroutine and communicate with one another via channels.

Fan Out

- `main.go` lines `154-156` fans out the files names to the workers that score the files.

This was done because while pipeline can help streamline some of the processing, fan in/fan out can be more effective if it contains a relatively time consuming task that would benefit from scaling up or being separate on its own thread.

Fan In

- `main.go` lines `33-56` fans in a varied number of channels into one output channel

Preventing Goroutine Locks

`done` channel first created at line `148` and used in `collectFileNames` , `gatherPaths` , `frameScorerWorker` , `fanInOutput` , `scorer.FilterAndMoveAllOutput` which helps it prevent goroutine locks by having it in the `select` statement where if can `case` on it like so:

```
select {
  case <-done:
    return
  ...
```

Both Functional and Data Decomposition Components Working Together

Line `151` in `main.go` is functional decomposition of file name manipulation. As the `MovieFrames` are coming in from line `151` in `main.go`, it is doing a data decomposition of parallelizing the scoring of the frames. After the frames are scored, they are sent for more functional decomposition of filtering the output. Thus function and data decomposition feed into one another and happen at the same time (while something is undergoing functional decomposition, another data decomposition of frames is happening somewhere else at the same time).