GPUC Freestyle Proposal - Motion interpolation

2019-12-17

My proposal is to implement a basic motion interpolation algorithm that mostly runs on the GPU.

1 Motivation

Almost all movies that exist are filmed at 24 frames per second. Most monitors run at 60Hz. The human eye can see up to at least 1000Hz. The low frame rate of movies has two problems:

(a) It's impossible to display video cleanly with a refresh rate that does not evenly divide the framerate. Usually extremely stupid techniques are used to solve this such as 3:2 pulldown [1] that result in visible "judder". Slightly better methods such as blending frames also exist [2] that somewhat remedy this.

(b) Even with solved judder, the video still stutters. This is especially noticeable when the camera pans.

(c) There is temporal aliasing since videos are usually filmed with a shutter angle of less than 180° which leads to actually wrong information such as tires rotating in the wrong direction.

Let's assume the original video material is fixed, i.e. we can't change what method was used to film the content. (c) is hard to solve without a more sophisticated algorithm (deeep leaaarning), so will be ignored here. (a) and (b) can both be solved by estimating the motion present in the scene, and resampling the video in the time domain using that motion. Ideally we would track this motion in 3D space, but that's far too much work so we just do everything in screenspace.

2 Related Work

There's lots of existing work for this. Many modern TVs implement this "Motion Boost" or "TruMotion" or similar. A PC program called SVP [3] also exists. There's also a lot of similar stuff integrated into Video editing software such as Adobe Premiere mostly for use in slow motion (that does not run in real time).

There is no existing program that has all components run on the GPU, and most existing software is proprietary.

3 Components

There are two separate parts to implement to make the basic version work.

3.1 Motion Estimation

We need to estimate the motion in the video. We restrict ourselves to only estimating two sets of motion vectors between two adjacent frames of the video (forwards and backwards). Using information from more previous or future frames should improve the result, but that's out of scope.

The probably simplest method to estimate motion is block-matching: For every block of pixels just search the neighborhood for the most similar block (e.g. sum of absolute differences). Blocks are not adjacent but overlapping by some percentage.

For optical flow in OpenCL there are some existing resources

- Pyramidal Lucas Kanade Optical Flow [4]
- (Intel: mostly optimization for an embedded system) [5]
- Classic-NL method implementation [6]
- OpenCV methods [7] . Based on Farneback [8]

Not sure if I actually want to bother with a sophisticated optical flow algorithm because it seems pretty complex. Need to do a bit more research. Probably just implement a block matching algorithm, that's what MVTools and SVP use with pretty good results. (or use an existing optical flow implementation).

3.2 Frame Rendering

This is mostly based on how MVTools [9] does it:

1. Take the left image and create black output image
2. For every block: Atomically add the color of the pixels to the same pixels shifted by the motion vector * delta time.
3. For every pixel: Normalize the pixel by the number of times it was written to
4. Do the same for the right image with backwards motion vectors
5. Blend the two resulting motion-compensated frames.

Not sure how they handle regions in the image that have no corresponding source pixels by this method. Might be completely prevented with the overlapping blocks but needs more research.

4 Additional Stuff

The above can already be done to a pretty complex level, but there's more stuff that could be done to make it usable.

4.1 Masking

Since there will be blocks / parts of the image that change between the frames, and just drawing those anyways will lead to artifacts, we might want to apply masking to the interpolated frame - if e.g. SAD is too much, we simply blend the frames without motion interpolation.

4.2 Making it work on video

The basic version would be to just have the program load two pictures and render the interpolated version between them, with a time parameter that can be adjusted. To make it more useful, it should load a video file and then actually iterate over the frames and do the algorithm. Depending on how important a nice looking demo is, I can adjust the priority of this.

4.3 Evaluation

Evaluating the results visually is probably enough, it should be pretty visible what artifacts we get and how usable it is. Alternatively, it would be possible to compare the resulting method with state of the art methods like [10], but probably not worth it since it's going to be really bad compared to that.

5 Points

20 points total

- 2 points: concept
- 14 points: implementation
  - 5 points: motion estimation using block matching in local memory
    - experiment with block sizes, overlap % and channels (chroma?)
    - implement naive / exhausting search
    - implement one of the optimized block matching algorithms mentioned in wikipedia
  - 5 points: frame rendering
    - apply forward and backward motion vectors to pixels of two images and blend them
    - 2 bonus points: masking the rendering at inconsistent locations to hide artifacts (needs research)
  - 3 points:
    - interactive ui that shows the motion vectors and transition between two video frames (with any t in [0,1])

        * loading images and converting to YUV space, storing them back to a file, and rendering a full video file
    - 1 point: performance tests, can we do real time?
- 2 points: presentation
- 2 points: q & a

6 reserach

hexagon search https://forum.doom9.org/showthread.php?p=693742

References

[1] "Three-two pull down," *Wikipedia*. May-2019 [Online]. Available: https://en.wikipedia.org/w/index.php?title=Three-two_pull_down&oldid=899459736. [Accessed: 17-Dec-2019]

[2] "mpv wiki interpolation." mpv, Dec-2019 [Online]. Available: https://github.com/mpv-player/mpv. [Accessed: 17-Dec-2019]

[3] "SVP – SmoothVideo Project – Real Time Video Frame Rate Conversion." [Online]. Available: https://www.svp-team.com/. [Accessed: 17-Dec-2019]

[4] J. Fung, "OpenCL Imaging on The GPU: Optical Flow." Mar-2011 [Online]. Available: https://www.khronos.org/assets/uploads/developers/library/2011_GDC_OpenCL/NVIDIA-OpenCL-Optical-Flow_GDC-Mar11.pdf

[5] D. Denisenko, "Lucas KanadeOptical Flow –from C to OpenCL on CV SoC." 08-Jul-2014 [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/support/examples/download/exm_opencl_lucas_kanade_optical_flow_with_opencl.pdf

[6] J. Moll, "OpenCL Implementation of the Classic+NL method for optical flow estimation: JesseMoll/CLFlow." Mar-2018 [Online]. Available: https://github.com/JesseMoll/CLFlow. [Accessed: 17-Dec-2019]

[7] "OpenCV: Optical Flow," *docs.opencv.org*. [Online]. Available: https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html. [Accessed: 17-Dec-2019]

[8] G. Farnebäck, "Two-Frame Motion Estimation Based on Polynomial Expansion," in *Image Analysis*, 2003, pp. 363–370 [Online]. Available: https://link.springer.com/chapter/10.1007/3-540-45103-X_50. [Accessed: 17-Dec-2019]

[9] "MVTools - Avisynth wiki," *avisynth.nl*. [Online]. Available: http://avisynth.nl/index.php/MVTools. [Accessed: 17-Dec-2019]

[10] H. Jiang, D. Sun, V. Jampani, M.-H. Yang, E. Learned-Miller, and J. Kautz, "Super SloMo: High Quality Estimation of Multiple Intermediate Frames for

Video Interpolation," *arXiv:1712.00080 [cs]*, Nov. 2017 [Online]. Available: http://arxiv.org/abs/1712.00080. [Accessed: 17-Dec-2019]