DSGA 3001 Final Project: Accelerated Video Face Detection and Filter Processing

Xintong Li, Yan Li, Zixuan Shao, Zichang Ye, Dian Zhang

1 Introduction

The world has recently seen a data explosion in image and video format due to better internet, mobile devices, and rapid expansion of video applications. From the perspectives of the machine learning and deep learning models, concise videos can take huge computing power and very long processing time, which leads to technical difficulties to achieve a real-time response. In this project, our group implemented multiple methods to accelerate applying filtering effects on the video clips.

2 Dataset

We used two video clips to test our baseline method as well as other acceleration methods. One clip named *hamilton_clip.mp4* came from face_recognition GitHub repository, and the other video clip named *peds* came from facenet_pytorch repository.

Hamilton_clip is an 80-second-long video clip of the musical Hamilton with 30 frames per second and has a total of 2356 frames, each frame with 360×640 pixel. In most of the frames, there is only one face. Video *peds*, filming pedestrians walking out of an opera house, has 105 frames, with 25 frames per second, so the length of this clip is 4 seconds. The height of each frame in *peds* is 1080 pixels, and the width is 1920 pixels. We chose it as the primary test data because it has a higher resolution and has more than one face in most frames, which helps us test the face detection process in a more complex environment.

To facilitate development, we created a shorter version for each of the two videos called *hamilton_short*, which has 834 frames and *peds_short*, which has 77 frames. We also created an extended version of *peds* by repeating it four times to test the scalability of our method while ensuring the same amount of faces in each frame of the longer video.

We used cv2.VideoCapture() to change an mp4 file into a 4d array (frames, height, width, channel) for later processes.

3 Methods

3.1 Face Detection

We explored two packages that detect faces in a video. The first package is the face_recognition[1] package that offers trained models such as SVM, k-NN, and CNN. This package is a handy and easy starting point but does not support GPU, making the usage of the deep learning model inefficient. We later discovered another face recognition package that implements Multi-Tasking Convolutional Neural Networks (MTCNN) in PyTorch [2], which allows us to make inference of a deep learning model on CUDA. Both methods take in a single frame and return a list of four coordinates that describe where the face locates.

3.2 Filtering Effects

We started by building a steady filtering process that applies a basic negative effect to the faces recognized. It is simply subtracting the current channel value of the Numpy arrays where faces are located from the maximum value (255 for RGB). After confirming the program is working correctly, we started adding other more complex filters.

Then, we used an oil effect filtering method to filter the recognized faces and implemented the algorithm with Numpy. This algorithm classifies all the surrounding pixels' average RGB channel values into intensity levels within the radius of the center pixel, and choose the intensity level with highest frequency, and average all pixels' RGB values within this intensity level as new RGB value for the center pixel. The last algorithm we use is the

[1] https://github.com/ageitgey/face_recognition

[2] https://github.com/timesler/facenet-pytorch

Figure 1: Original image, and image after negative effect, oil effect and blurring effect

Figure 2: Snapshot of the Processed Video

standard box filters that iterate over the target pixels and take an average of an odd square convolution as the new pixel of the center pixel for the new image array.

For both algorithms, the radius of convolution enhances the filtering effect of the face. On a single frame, the baseline implementation of the box filter is about 50 times faster than the oil effect filter, so we chose the box filter as the filtering method in video processing for the sake of algorithm speed and focused on accelerating this box filter.

A snapshot of the outcome of face detection and blurring effect is presented in Figure 2.

3.3 Optimization

Our efforts in optimization mainly focus on accelerating the blurring effects on detected faces.

3.3.1 Baseline

We combined the CPU version of the face detection feature from the Facenet package with the Pytorch backend and naive Numpy implementation of the box filtering algorithm described in Section 3.2 as our baseline for optimization.

3.3.2 Scipy Convolution

Blurring an image is equivalent to a convolution operation with a $n \times n$ kernel of all ones, multiplied a factor of $1/(n^2)$, where n is the width and height of the neighborhood square. We used the function scipy.signal.convolve2d() to perform the neighborhood averaging for each color channel, therefore avoiding the double loop in our baseline implementation.

3.3.3 Cython and Numba

We tried both Cython and Numba to accelerate the box blurring algorithm. For Cython, we attempted to pre-define the major variables and arrays as ctype, such as each image frame's array and the detected faces array, but no significant acceleration has been observed. Two obstacles potentially make the application of Cython not as promising as we would like:

- The detected faces array is not of constant shape, especially when there is no face found or numerous faces recognized;
- Cython requires pre-definition of the data type in arrays. At the same time, different recognition and image output packages use different integer formats, and thus the data type needs to be cast frequently, which can potentially slow things down.

For Numba, we also tried the parallelization feature of JIT to parallelize the image processing. There is also not much speedup on a single image, but it makes the whole process five times faster on the 50 frames video Numpy array. The main issue with applying Numba across our entire solution is that Numba requires the classes, and methods that are used are also jit-decorated, but our MTCNN modules are not jit-decorated. Therefore we used turned to the Multiprocessing module instead to implement parallelization.

3.3.4 Multiprocessing

We tried to attain concurrency by implementing the multiprocessing with processes and queues. We first divided the video array into N indexed smaller arrays. Then the blurring filter is applied to each smaller array in N processes in parallel. Finally, we collected the results from these processes in a queue and sorted them by index to restore order. This solution further accelerated the solution with

convolution by 14% when $N = 4$, and has no significant speed up with more processes.

3.3.5 CUDA Programming

In general, GPU has natural advantages over CPU on image processing algorithms. We implemented a Numba Cuda decorated function that could write a blurred array based on the original Numpy array representing a detected face in one frame. To better accommodate the characteristic of 2d image, we choose both block dimension and grid dimension to be two-dimension so that each pixel is one thread in the architecture. We choose the standard (32, 32) for the number of threads per block, and block size per grid is calculated by the division of original array size and threads per block. In the CUDA function, we also pass in arguments such as normalization term and radius to speed up the calculation unit. The rest of the implementation is similar to baseline implementation, but more control flows are needed to handle the bound of the array.

4 Results

4.1 Time Comparisons

We obtained a 6x speedup with our CUDA solution compared to our initial solution and performed similarly to the blur() method provided by OpenCV, which implements the convolution operations in C++. The result is presented in Figure 3[3]. Note that cv2.blur() only uses CPU for the convolution operations. The efficiency of cv2.blur() comes from insights beyond computational optimizations. For example, when the kernel is separable, it can split the kernel into two smaller kernels and reduces the number of operations. In addition, it takes advantage of the Convolution Theorem[4] which reduces the time complexity of naive convolution method from $O(n^2)$ to $O(nlog(n))$[5].

4.2 Decomposition of the Processing Time

We used line profiler to check the processing time for each step of the execution, specifically the face detection step and the blur filtering step, and the results for the video peds.mp4 are in Table 1. Using GPU sped up the face detection step significantly

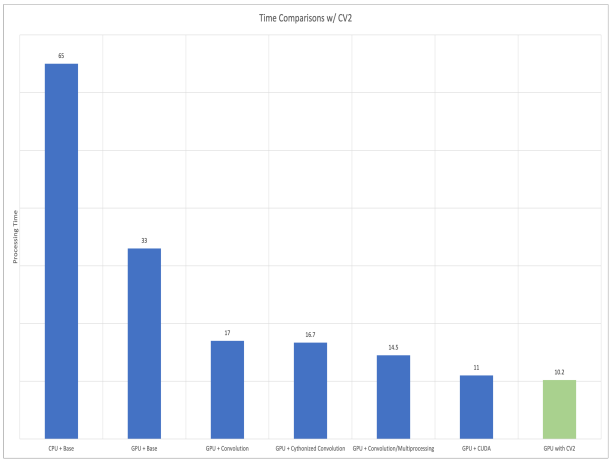

Figure 3: Time Comparisons of Different Solutions

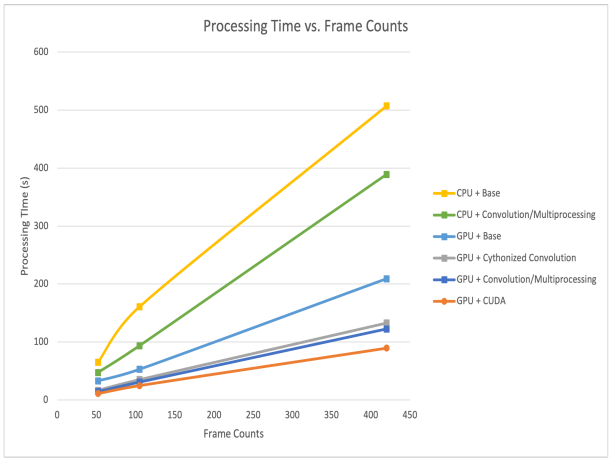

Figure 4: Processing Time vs. Frame Counts

comparing to the CPU. And the scipy convolution operation and Cuda programming both resulted in significant improvement in the processing time of the blur filtering step. Interestingly, even though the solution with multiprocessing is slightly faster than the scipy solution in the time comparisons, we are not observing that improvement when we line-profiled the multiprocessing blurred filtering. We researched and found that line profiler may need additional efforts when dealing with multiprocessing, as it may only profile the main process.

4.3 Scalability

In Figure 4, we tested our different solutions to videos of increasing size. We found that our method that utilizes GPU for both face detection and filtering is the most scalable one, in the sense that its run time grows relatively slow. On the other hand, if GPU resource is not available at all, the run time of our best-optimized CPU solution grows very fast at the increasing frames, making the current method very inefficient.

[3]Please note that in our figures, the name of a solution means "hardware for face detection + technique for filtering." For example, "GPU + Cythonized Convolution" represents that we used GPU for model inference and Cythonized Convolution for blurring effects.

[4]https://en.wikipedia.org/wiki/Convolution_theorem

[5]https://stackoverflow.com/questions/31336186/opencv-computational-efficiency-of-filter2d-function

Methods	Face detection	Blur filtering
CPU+Base	136.3s	42.5s
GPU+Base	37.6s	41.2s
GPU+Convolution	24.9s	12.8s
GPU+Cython Convo	25.3s	12.8s
GPU+Numba Convo	25.2s	12.7s
GPU+Multiprocessing Convo	24.8s	15.2s
GPU+CUDA	25.0s	2.2s

Table 1: Time Comparisons of Different Solutions on Different Steps for video peds.mp4

5 Discussion & Next Steps

In this project, we did not focus on speeding up the inference of deep learning models besides the usage of PyTorch. However, the inference part of the problem may also be accelerated with the multiprocessing module of torch, as well as usage of other smaller models. Suppose GPU is not available, since the inference of neural network is inefficient on CPU, we would recommend switching to a smaller yet less powerful model, such as SVM to ensure of the usability of the program.
Memory limitation can also be an issue as video can easily be minutes long. If the video is too huge to fit into the memory, we could read and process the video frame by frame instead of directly converting it to a 4d Numpy array. We are then required to find an efficient way to store the processed results of each frame.
While "plug-in" methods such as Cython and Numba can be very helpful, the state-of-the-art optimization often comes from a deeper understanding of the original problem itself. The application of the mathematical properties and theorems of the convolution operation that reduces the necessary computation in OpenCV is a great example.

6 Reference

- Tim Esler's implementation of MTCNN in PyTorch: https://github.com/timesler/facenet-pytorch
- Adam Geitgey 's implementation of several trained face detection models, which are often more efficient for CPUs: https://github.com/ageitgey/face_recognition
- Why CV2 is fast: https://stackoverflow.com/questions/31336186/opencv-computational-efficiency-of-filter2d-function
- K. Zhang, Z. Zhang, Z. Li and Y. Qiao. Joint Face Detection and Alignment Using Multi-task Cascaded Convolutional Networks, IEEE Signal Processing Letters, 2016.