ECE 1782 Final Project Report

Topic: Video Block Matching for Compression

Runheng Lu

Yiming Zheng

Jingsheng Zhang

Dengsong Wang

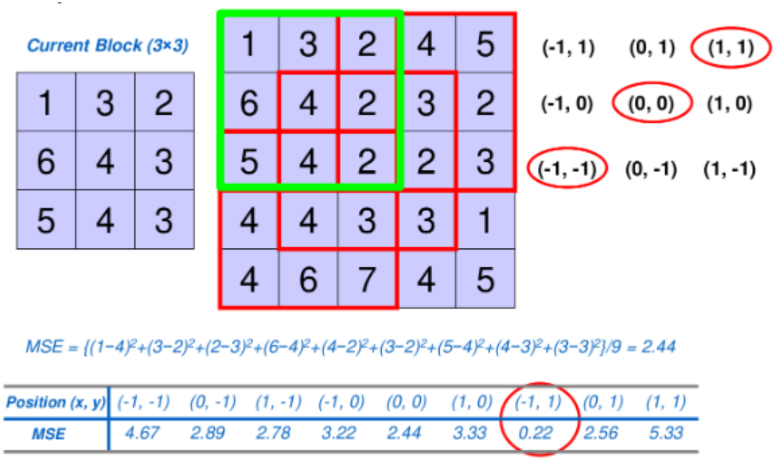
1. Introduction

1.1 Problem statement

Our group decided to use the CUDA program to accelerate video encoding speed.

Video encoders were invented to compress video data. We will use a lossy video compression method. By reducing the quality of the frame images in the video, the amount of data is significantly reduced, and the human eye cannot distinguish the difference between the frame images before and after compression. H.264, one of the most widely used video coding standards, can achieve compression ratios of about 50:1 to 100:1 and the peak signal-to-noise ratio (PSNR), which describes image quality of about 37 dB to 40 dB, which means that the compressed video maintains very high image quality while significantly conserving storage space. It mainly uses these techniques to compress video data:

- H.264 describes the original video in two files: an MV (motion vector) file and a residual file (the Difference between prediction and original data).
- H.264 divides frames into I (intra) and P (predicted) frames. Due to the similarity of images between adjacent frames, each block in a P frame (usually 8×8 in size) can use a block from the previous frame as a base and just record the difference. In most cases, we can find a very similar block.
- H.264 uses ME (motion estimation) to find the most similar block for each block. As in Figure 1, the block with the smallest MSE (mean-square error) or SAD (sum of absolute differences), the most similar block from a range of the reference frame, will be found as the basis for constructing the current block. As in Figure 2, The MV file records the relative position of the reference block and the current block as a motion vector.



$$MSE = \{(1-4)^2+(3-2)^2+(2-3)^2+(6-4)^2+(4-2)^2+(3-2)^2+(5-4)^2+(4-3)^2+(3-3)^2\}/9 = 2.44$$

Position (x, y)	(-1, -1)	(0, -1)	(1, -1)	(-1, 0)	(0, 0)	(1, 0)	(-1, 1)	(0, 1)	(1, 1)
MSE	4.67	2.89	2.78	3.22	2.44	3.33	0.22	2.56	5.33

Figure 1

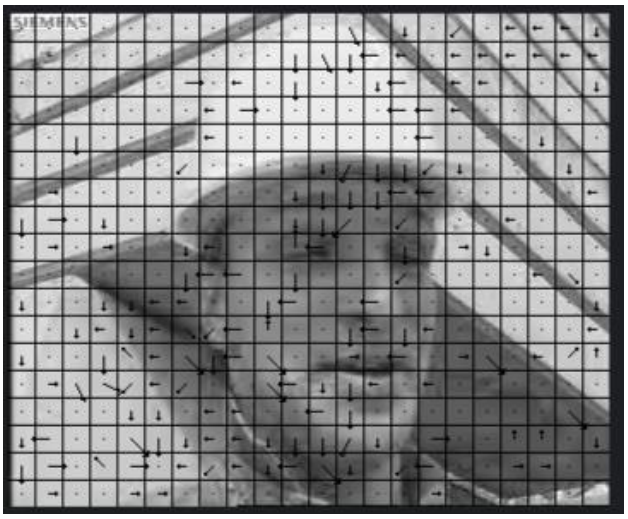


Figure 2

- For the difference between the reference block and the current block, H.264 uses DCT (Discrete Cosine Transform) and quantization to encode the data, compress it using differential encoding and entropy encoding, and then store it in a residual file. During this process, the data size is reduced, and quality loss happens.

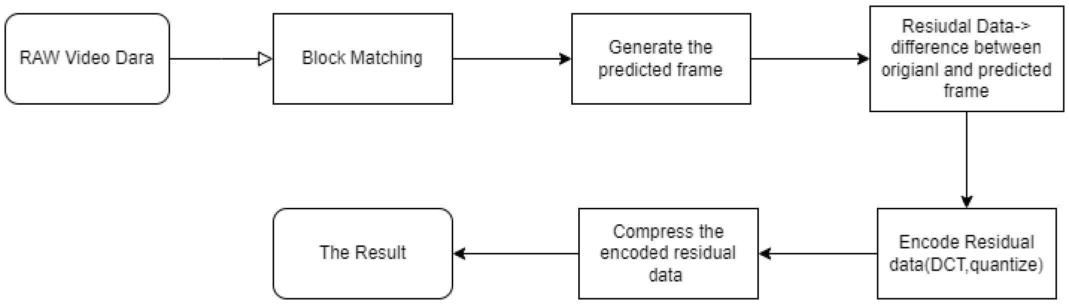

RAW Video Dara
Block Matching
Generate the predicted frame
Resiudal Data-> difference between origianl and predicted frame
Encode Residual data(DCT,quantize)
Compress the encoded residual data
The Result

Figure 3: Flow chart of video encoding progress

Our project aims to parallelize the video encoding process. After optimization, compare the processing speed of the parallelized version with the version calculated using CPU and the version using the existing video encoding tool FFmpeg.

1.2 Motivation

Motion estimation, specifically block matching, is a critical process in video compression. It is the most time-consuming aspect of video encoding. For instance, for a 720p video (1280×720), if we set the block size to 8×8 and the search range r = 8, each frame needs to calculate about 4.16 million SAD (Sum of Absolute Differences) to find the most similar block in the previous frame for each block in the current frame. A 10-minute video with a frame rate of 30fps needs approximately 74.9 billion SAD calculations. Since many blocks are parallelizable, GPU parallel computation can speed it up.

Motion estimation can be conducted simultaneously because the process involves analyzing different blocks of an image independently of each other. In video compression, each frame is divided into blocks. The task is to compare each block from a current frame to various blocks in the previous frames to track movements. Since the analysis of each block is independent, the calculation for one block doesn't depend on the results from another. Therefore, these comparisons can be made concurrently. This simultaneous processing is what makes motion estimation highly suitable for parallel computing platforms like GPUs. It will greatly speed up the overall process.

2. Overview of our solution

Our solution uses the CUDA language and contains two .cu files. The main cu file contains the main program and kernel function, and the util cu file contains defined constants, basic program functions, and time measurement functions.

During the optimization of the GPU program, our project went through two iterations using different GPU optimization approaches. The first approach was based on optimizing by applying CUDA threads, and we added padding for each frame and transferred global memory into shared memory to increase GPU running speed. The second approach is the version that we eventually applied. Different from approach one, we found that the CUDA Block-Per-Block Motion Vector Matching is more optimal for motion estimation than the CUDA Thread-Per-Block Matching.

2.1 Preparation Works

First, we wrote a block-matching program entirely based on the CPU, and then we developed the GPU version based on this CPU version. To begin with the GPU version, to verify the kernel function's correctness and reduce the difficulty of debugging, we initially processed only one frame to confirm whether the kernel function correctly provided the motion vector for the corresponding block.

For the block matching on a single frame, to check if the kernel function can correctly give the output, we will compare the output of the motion vectors for the same frame between the GPU and CPU versions. There are four essential functions, including block matching and calculating SAD (Sum of Absolute Difference). First, in the main function, we allocated the necessary memory on the host and device for the current frame, reference frame, motion vectors, and SAD list. Then, we copied the current and reference frames data from the host to the device and set up appropriate block sizes and grid dimensions. Following that, we called the kernel function for block matching using the diamond search algorithm. The parameters passed were the current frame, reference frame, motion vectors, and SAD list. Inside the kernel function, we get the best SAD by calling the function to calculate SAD. Based on the lowest SAD, we determined the motion vectors for the current block in the x and y directions. We repeated this step to calculate the motion vectors for all blocks in the current frame. Then, we compared these motion vectors with those outputs from the CPU side. If the results matched, it confirmed that our kernel function was correct. Thus, in the subsequent complete GPU-side code, we would not need to verify the accuracy of the kernel function again.

2.2 Approach 1: CUDA Thread-Per-Block Matching

2.2.1 Simple GPU Approach

1) **kernel operation:**

   Input data: The first frame of each stream is used as the current frame and reference frame. For the other frames in each stream, the current frame is used as the current frame, and the output temp frame of the previous frame is used as the reference frame.

   Output data: reconstructed frame.

   Process:

   - Block match: We used the full search strategy. A search of 17 × 17 motion vectors is performed for each block. Each time the search is performed, calculate the SAD of the target block and the current block. The block with the smallest SAD is taken as the final target block, and the motion vector between it and the current is recorded.

   - Predict: For each pixel, the source pixel position in the reference frame is obtained by adding the current pixel position to the best motion vector. Then, the pixel value of the source pixel position in the reference frame is assigned to the current pixel position in the temp frame, and output the temp frame.

2) **Stream operation**: The number of frames we process per stream is set to GOP, and its value is 15. In this version, the video has a total of 300 frames, so it is divided into 300/15=20 streams.

2.2.2 Add Padding

When searching for the best motion vector in the kernel, if a block is at the edge of the frame, an “if” judgment is needed to prevent the search area from exceeding the frame edge. In order to prevent this “if” judgment to further speed up the kernel running speed, we first try to add an input named “d_ref_pad_frame” when calling the kernel. “d_ref_pad_frame” is initialized to a frame whose length and width are 8 (equal to the motion vector search radius) more than the original frame, and all its pixel brightness is set to the lowest value. Inside the kernel, replace the center part of “d_ref_pad_frame” except the outermost 8 pixels width with the data of the reference frame, making it equivalent to the reference frame periphery plus padding with a width of 8. Use “d_ref_pad_frame” in the kernel instead of the reference frame to search for motion vectors and generate predicted blocks. The total number of blocks and block size in each frame does not change, but the x and y coordinates of each pixel are added by 8 to the values before adding padding. The image of the original frame is still divided correctly, and the padding area is just used to avoid the problem of out-of-range error.

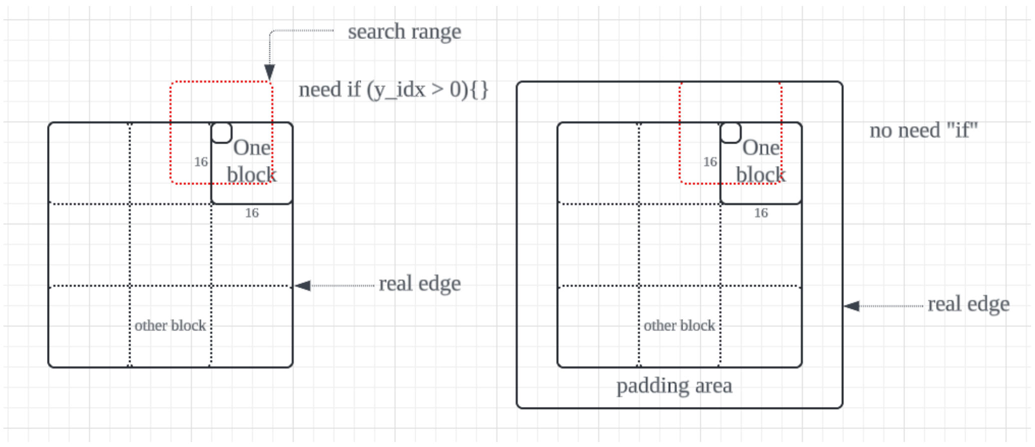


Figure 4

2.2.3 Advanced Padding

The other version of adding padding: when we read the video file at the beginning of the project, we added a circle of width equal to the motion vector search radius, which is 8 pixels of padding, to the periphery of all frames. The brightness value of padding pixels is set to the highest value. In this version, the length and width of all frames in this project have increased by 8, but only the center of the frame without padding area will be divided into blocks. Block size and number of blocks remain unchanged.

2.2.4 Shared Memory

We defined shared memory inside the kernel function. In our kernel function, we defined shared memory for the current and reference frames. The size of the shared current frame is defined as block size by block size, and due to the addition of padding, the size of the shared reference frame is defined as [block size + 2 Padding] by [block size + 2 Padding]. Next, we perform boundary checks to ensure the current index is within the block size. If it is within the range, we calculate the global memory location corresponding to the current thread, then read data from the current and reference frames in the global memory and store it in the corresponding shared memory location. Lastly, we use "__syncthreads()" to ensure all threads have synchronized their data loading. Then, we modify the calculating SAD function to ensure it uses parameters from the shared memory. However, after we added the shared memory, the output for the reconstructed frame will have errors. We will discuss this problem in the discussion part.

2.3 Approach 2: CUDA Block-Per-Block Motion Vector Matching

In the first approach, we found that assigning one thread too much work causes an imbalance in memory space and thread capacity. So, we also implemented another approach using more CUDA threads to encode a frame block.

2.3.1 Simple Approach

Within this approach, we assign the work of motion estimation of a video block into a CUDA block where each thread in that block calculates the MAE for a possible motion vector. In our full search algorithm with search_range = 8, we need to compare 289 blocks and choose the smallest motion vector.

In this approach, we let each thread calculate the SAD of one motion vector. As a CUDA block contains all 289 threads, we can use regression to find the smallest motion vector.

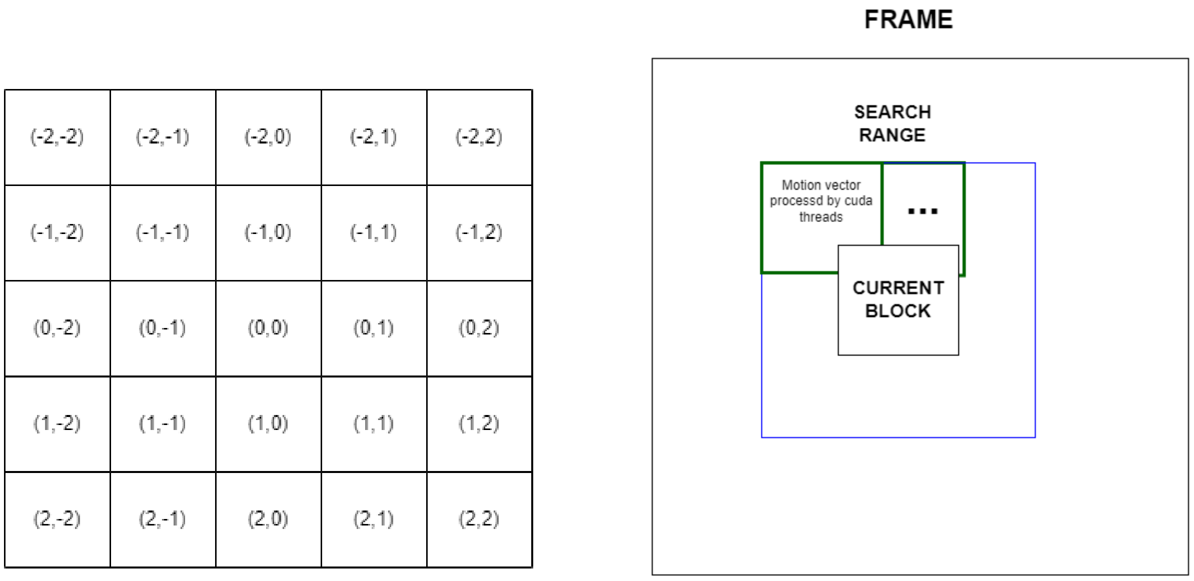


Fig 5. Motion Vector, and Approach 2 distribution

2.3.2 Using Shared Memory

In our simple block-per-block approach, we only use shared memory for regression. In this approach, we further load the reference frame and current frame (both used to estimate motion) to accelerate the SAD calculation in each thread.

2.3.3 Optimize the Regression Algorithm

The next step is to optimize the calculation of minimal MAE. We use the reduction method we learned in the lecture (reduction #2, lecture 7). We fix the branch divergence by transferring the step size into ½ current level, so all the calculations in extracting minimal MAE can be done in the previous threads.

3. Evaluation

3.1 The Best Multi-core CPU-based Solution: ffmpeg

FFmpeg (Fast Forward Moving Picture Experts Group) is an open-source project for processing multimedia. The most popular function is a command-line tool called “ffmpeg” which is mainly used for video and audio encoding and decoding. Its main features are fast running speed, high output quality, and small file size. Figure 5 shows the parallel CPU encoding process in ffmpeg.

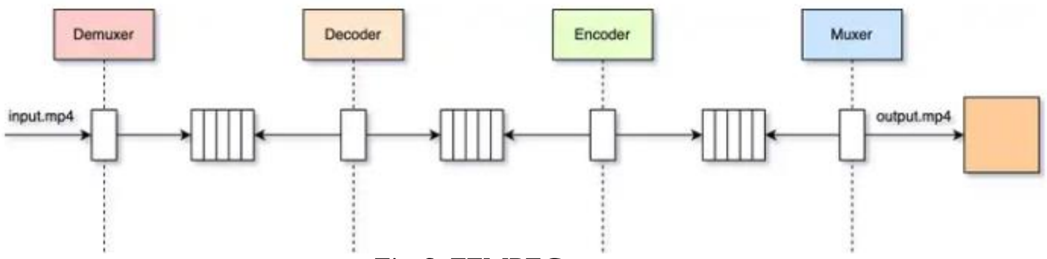
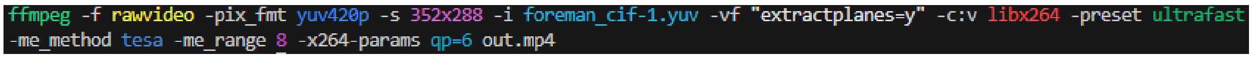


Fig 6. FFMPEG process

In order to compare our approaches to the FFmpeg we use the following parameter:



```
ffmpeg -f rawvideo -pix_fmt yuv420p -s 352x288 -i foreman_cif-1.yuv -vf "extractplanes=y" -c:v libx264 -preset ultrafast
-me_method tesa -me_range 8 -x264-params qp=6 out.mp4
```

In this case, we set the parameters (search range, block size, frame size, prediction mode) in the same way as our model and set the encoding present to ultrafast to minimize the influence of the compression time during encoding.

3.2 Experiments and Results

3.2.1 Experiments Settings

For the CPU we use i5-12600K with 32GB RAM. For the GPU we use the video cards from the lab (RTX3070).

For our five major models, we have tested their performance under different CUDA configurations.

3.2.2 Experiments of our 5 Models

1) Simple Thread-Per-Block Matching (approach 2.2.1): we tested the results using different grid sizes and block sizes. The result is shown in table 1.

Block Size	Time (s)	Block Size	Time (s)
1×1	4.1996	8×8	4.3261
2×2	2.3735	16×16	27.1602
4×4	2.6690		

Table 1

We can see that a low video block size can speed up the performance. This might be due to the complexity of the work for each thread. However, a low block size also causes potential resource waste. The 2*2 gird gives the best performance in this case.

2) Thread-Per-Block Matching with Thread Padding(approach 2.2.2):

Block Size	Time (s)	Block Size	Time (s)
1×1	9.6106	8×8	8.5939
2×2	2.4988	16×16	11.9260
4×4	2.7773		

Table 2

Padding in each thread does not speed up the performance in small block sizes, and it has optimized the performance in large block sizes. The acceleration in large block size might be because of poor memory usage in the original approach, and this padding approach solved this problem.

3) Thread-Per-Block Matching with Initial Memory Padding (approach 2.2.3):

Block Size	Time (s)	Block Size	Time (s)
1×1	4.1204	8×8	6.3451
2×2	1.4238	16×16	10.7772
4×4	3.1310		

Table 3

In this version, we optimize the padding approach by moving it to the frame reading stage. We reach the best result of approach 1 with block size 2×2. By adding padding while reading the frame data, we avoid adding tasks to each GPU thread. And thus gained the benefits of using padding, and released the drawback of accessing and transforming larger data.

4) For our CUDA Block-Per-Block Matching (approach 2), the block is fixed in this approach, so we don’t need to compare the differences when using different CUDA configurations. We listed the results Table 4.

   For our straightforward approach, we achieved 0.2653s. This means using full search as our search algorithm (approach 2.3.1) is far better than approach 1. The reason for this might be that we balanced each CUDA thread’s computational power in a reasonable amount, and the memory became more coalesced.

CUDA Block-Per-Block Matching with SMEM (approach 2.3.2) uses 0.4709s, which means adding SMEM makes our architecture worse. This might be because we already used shared memory in regression. Adding frame data to SMEM caused conflict.

In approach 2.3.3, we optimized the regression algorithm and avoided conflict in shared memory. And this is our best approach, causing 0.2482s.

Block Size	Time (s)
simple CUDA Block-Per-Block Matching (2.3.1)	0.2653s
CUDA Block-Per-Block Matching with SMEM (2.3.2)	0.4709
Optimized Regression CUDA Block-Per-Block Matching (2.3.3)	0.2482

Table 4

3.2.3 Comparison of the Best Results

We have listed our best results for all approaches and the result of FFmpeg in Table 5. Approach 1 reached 1.85 times speed up, and approach 2 reached 10 times speed up.

Algorithm	Time (s)
FFmpeg	2.6435
Simple Thread-Per-Block Matching (2×2)	2.3735
Padded Thread-Per-Block Matching (2×2)	1.4238
CUDA Block-Per-Block Matching (2.3.3)	0.2653

Table 5

3.3 Analysis

In conclusion, our second approach has achieved significantly better results compared to our first approach. The likely reason for this improvement is that our second approach did a better job in work assignments. Each thread in the second approach has a lot fewer tasks compared to the previous approach. This results in a more efficient use of GPU resources. This has also led to a more reasonable memory access pattern, further enhancing performance.

4. Discussion

4.1 Conclusion

Generally speaking, we have achieved an optimal result with up to 10 times improvement compared to the best CPU parallel algorithm Ffmpeg. During this project, we have learned a lot about parallel CUDA programming, and here are our thoughts:

4.2 Insights

4.2.1 Change Thread Level into Block Level Parallelism in Approach 2

Our approach 2 uses block-level parallelism, and we have found that it performs optimally when dealing with short videos with small resolutions. This is likely due to the fact that it successfully divides the work and can efficiently utilize the GPU resource.

4.2.2 Potential Issues of Adding Shared Memory in Approach 1

As section 2.2.3 mentioned, after we add shared memory in the kernel function, the reconstructed frame will be different from using global memory for the current frame and reference frame. One potential problem is the step of loading data into shared memory. We use a dimensioned array to store the Y components of each frame, including the current and reference frames. When we define the shared memory, we create two-dimensional arrays for each frame. Therefore, when we load data from one dimension array into a two-dimensional array, the data cannot be loaded into the correct index due to the incorrect definition of the current and reference frame's indexes.

4.3 Further Improvements

4.3.1 I-Frame Construction

The I-frame (intra-coded frame) is self-contained and does not have a dependency on other previous frames. Since our example YUV file has few scene changes, we did not consider encoding i-frames. Constructing an I-frame will improve the GPU’s processing performance if the video contains numerous scene changes. We may consider adding I-frames in future work.

4.3.2 Transformation and Quantification

Regarding the Discrete Cosine Transform (DCT), we have implemented only basic optimization techniques using the most straightforward method for computation. Future optimization could involve using FFT (Fast Fourier Transform)-based DCT and IDCT. For quantization, our Quantize and Rescale functions include some calculations that could be pre-computed and stored in a lookup table, such as the values of d_qMatrix. This approach would reduce GPU computational load during runtime.

4.3.3 Other Searching Algorithms

In our project, we only use full search as our search algorithm, but this is a very slow algorithm, and modern video encoders seldom use it in motion estimation. We complete a diamond search algorithm, but its logic is very different from a full search, making it very hard to implement these different search algorithms simultaneously.
So, for further improvements, we want to implement and optimize diamond search and try to apply it together with full search so that users can select them by their choice.

5. Related Work

For the related work, in CUDA Memory Optimisation Strategies for motion estimation, the author uses a similar method to our approach 2, and they successfully achieve up to 50 times faster than CPU implementations. Their result seems slightly better than ours. This might be because their experiment used an old CPU with only 8G RAM, and they did not clarify the CPU motion estimation used, so it might not be as powerful as ffmpeg.

6. Contributions

Dengsong Wang: Appoarch 1 solution design, Optimization

Jingsheng Zhang: FFMPEG (CPU) compare, Optimization SMEM

Runheng Lu: Approach 1 solution, Optimization Stream

Yiming Zheng: Approach 2 solution design, Optimization

Reference

Sayadi, F. E., Chouchene, M., Bahri, H., Khemiri, R., & Atri, M. (2018). Cuda Memory Optimisation Strategies for motion estimation. *IET Computers &amp; Digital Techniques*, *13*(1), 20–27. https://doi.org/10.1049/iet-cdt.2017.0149

FFmpeg. (n.d.). https://ffmpeg.org/