A Multithreaded Robust C/C++ Implementation of Video Data Compression

Tong Lai Yu
School of Computer Science and Engineering
California State University,
San Bernardino
tyu@csusb.edu

WORLDCOMP’17:IPCV17, July 17-20, 2017, Las Vegas, Nevada, USA.

Keywords: Video, Data Compression, Producer-Consumer, Condition Variables, Multi-threads

Abstract

A producer-consumer paradigm is used to implement an MPEG-4-like video encoder-decoder (codec). The paradigm often used to synchronize events in a computing system naturally fits the pipeline architecture of such a codec. Each stage is processed by a thread, which acts as a producer and its outputs feed to the next stage handled by another thread, which acts as a consumer of the previous stage as well as a producer for the next stage. Condition variables, which are containers of threads that are waiting for certain conditions to be met, are used to ensure the proper access to the buffered data passing from producers to consumers.

1. Introduction

In the past two decades as the Internet grows exponentially, multimedia applications become ubiquitous. Data compression is the soul of the engine that drives the rapid development of these applications. Audio and image data can be effectively transmitted across the Web or saved in a digital storage medium (DSM) only after they have been compressed. Video compression can be considered as an extension of audio and image compression and in recent years video compression products have experienced rapid growth in a variety of consumer products such as iPods, iTune, mobile phones, digital cameras, TV games, and many kinds of hand-held devices.

To achieve high compression efficiency, compression techniques often exploit the inherent temporal, or time-based redundancies between frames. Encoding each frame individually, which does not exploit any temporal redundancy is referred to as intra-frame coding, where at a certain instance, the codec applies data processing only to the data of the current frame but not to any other frame in the video sequence. This contrasts with inter-frame coding, where processing is applied simultaneously to the data of the current frame and the adjacent frames to exploit temporal redundancies.

Compression standards such as MPEG-4, introduced in the late 1990s include compression of AV data for web (streaming media) and CD distribution, voice (telephone, videophone) and broadcast television applications. The standards refer the intra-frames to as I-frames and call the inter-frames P-frames. P here refers to ‘prediction’. This is because people use motion estimation (ME) and a technique known as block-based motion compensated prediction to exploit the temporal correlations between frames in order to reduce temporal redundancies.

Figure 1a presents a typical video data encoder and Figure 1b shows a corresponding decoder. In the figures, the terms DCT, MV, ME, MC, and $Q^{-1}$ represent discrete cosine transform, motion vector, motion estimation, motion compensation, and inverse quantization respectively. As shown in Figure 1a, encoding of a frame consists of a forward path and a reconstruction path. The goal of the reconstruction process is to obtain a reference frame for motion compensations of the macroblocks of the next frame. Many of the reconstruction functions are also used in the decoding process. The following shows the encoding steps:

Encoding:

1. Forward Encoding:
   1. Read a 24-bit RGB image frame from an uncompressed avi file.
   2. Decompose the RGB frame into $16 \times 16$ macroblocks.
   3. Transform and down-sample each $16 \times 16$ RGB macroblock to six $8 \times 8$ YCbCr sample blocks using YCbCr 4:2:0 format.
   4. Search the reference frame for a ‘best-matched’ YCbCr macroblock, which we call it reference macroblock.
   5. Encode the motion vector (MV) of the reference macroblock found using pre-calculated Huffman codes and transmit the codewords.
   6. Calculate the residuals between the current macroblock and the reference macroblock.
7. Apply Discrete Cosine Transform (DCT) to the residuals.

8. Forward-quantize the DCT block; reconstruct the block as described in reconstruction path.

9. Reorder each quantized $8 \times 8$ DCT block in a zigzag manner.

10. Run-level encode each quantized reordered DCT block to obtain 3D (run, level, last) tuples.

11. Use pre-calculated Huffman codewords along with sign bits to encode the 3D tuples.

12. Transmit the codewords.

2. Reconstruction Path:

1. Inverse-quantize each quantized DCT block. (Note that quantization is a lossy process and therefore, the recovered block is not identical to the one before quantization.)

2. Apply Inverse DCT (IDCT) to the resulted DCT blocks to obtain $8 \times 8$ YCbCr residual sample blocks.

3. Add the residual sample blocks to the corresponding reference sample blocks that were previously encoded and reconstructed to obtain the reference macroblock.

4. Save the reconstructed reference macroblock in a vector which will be used to construct one frame.

5. Convert each of the YCbCr macroblock saved in the vector to an RGB macroblock, which is then saved in a buffer. The buffer is the reconstructed frame.

Though there is a large volume of research papers on improving and analyzing video data compression\[4, 6, 14\], there are very few papers discussing the implementation of it. In this article, we focus on presenting a multi-threaded C/C++ implementation that exhibits robustness and flexibilities.

2. Tasks Synchronization

The encoding (and decoding) process can be divided into a number of tasks. At a very coarse level, it consists of the task of getting the input data stream, the task of encoding the data stream, and the task of playing the video frames. (The encoding task can be further divided into sub-tasks.) To avoid the tasks from interfering each other, and from tangling into a complex large job, we let each crucial task run in its own thread independently, and employ the producer-consumer paradigm to synchronize the tasks. The producer-consumer paradigm\[7, 9\] is a well-studied synchronization problem in Computer Science. A classic producer-consumer problem has two threads (one called the producer, the other the consumer) sharing a common bounded buffer. The producer inserts data into the buffer, and the consumer takes the data out. In our case, the buffer is a queue where data are entered at the tail and are read at the head. Physically, the queue is a circular queue. Logically, one can imagine it to be a linear infinite queue\[10\]. The head and tail pointers are always advancing (incrementing) to the right. (To access a buffer location, the pointer is always taken the mod of the physical queue length, e.g. $\text{tail} \% \text{queue.length}$.) If the head pointer catches up with the tail pointer (i.e. $\text{head} = \text{tail}$), the queue is empty, and the consumer must wait. If the difference between the head and the tail is equal to the length of the buffer, the queue is full, and the producer must wait.

Figure 2 shows a variation of the traditional producer-consumer problem that we can use to synchronize the encoding tasks. In the figure, the $\text{dataSource}$ thread acts as a producer that gets data from a data source and puts the data in two independent shared buffers, $\text{buffer 1}$ and $\text{buffer 2}$. 

Decoding:

1. Construct a Huffman tree from pre-calculated Huffman codewords for decoding 3D run-level tuples.

2. Construct a Huffman tree from pre-calculated Huffman codewords for decoding motions vectors (MVs).

3. Read the bits of the bit stream from the encoded file and traverse the MV Huffman tree to recover the TSS MVs of a macroblock.

4. Read the bits from the encoded file and traverse the 3D run-level Huffman tree to recover 3D run-level tuples to obtain $8 \times 8$ DCT blocks.

5. Reverse-reorder and inverse-quantize each DCT block.

6. Apply Inverse DCT (IDCT) to the resulted DCT blocks of Step 5 to obtain $8 \times 8$ YCbCr residual sample blocks.

7. Add the residual sample blocks to the corresponding reference sample blocks that were previously encoded and reconstructed frame.

8. Save the reconstructed reference macroblock in a vector, which will be used to construct one frame.

9. Convert each of the YCbCr macroblock saved in the vector to an RGB macroblock, which is then saved in a buffer. The buffer is the reconstructed frame.
The encoder thread encodes the data while the player thread performs the play-back function. The encoder thread and the player thread are consumers, fetching data from buffer 1 and buffer 2 respectively. If buffer 1 is empty, encoder sleeps. If buffer 2 is empty, player sleeps. If either of buffer 1 or buffer 2 is full, the dataSource thread has to sleep. Consequently, encoder cannot run a lot faster than player. If one needs fast encoding, the play-back mechanism has to be turned off. The encoder thread can also act as a producer, placing the compressed data in another buffer for another consumer thread to retrieve the data for storage or transmission. Note that a consumer does not need to delete the data. It simply advances its head pointer after reading the data. The producer overwrites the old values with new data.

Each slot in a buffer can hold a Frame object that contains the data and some relevant information of a video frame. We can conveniently declare a Frame class like the following code segment, which has omitted some minor details for clarity of presentation:

```cpp
class Frame {
private:
   // number of data bytes in a video frame
   int frameSize;
public:
   // pointer to memory storing a frame data
   char *buf;
   // constructors
   Frame ();
   Frame ( int frameSize0 );
   // read one frame from data source
   int readFrame ( FILE *input );
   // destructor
   ~Frame();
};
```

The memory holding a frame of video data can be allocated in a constructor like the following code:

```cpp
frame::Frame ( int frameSize0 )
{
   frameSize = frameSize0;
   buf = ( char * ) malloc ( frameSize );
   assert ( buf );
}
```

---

3. Condition Variables

In a multi-threaded program, it is common for a thread to wait for some condition to become true before proceeding. An efficient way to handle waiting for conditions is to use condition variables[5], which are useful and popular tools for synchronizing events in a computing system. We use them to ensure that data are accessed properly.

A condition variable is an explicit queue that a thread can put itself in when the thread has to wait for some condition to become true. The thread sleeps in the queue; another thread could change the condition to true and signal the sleeping thread to wake up. The idea was mostly formatted by Hoare in his work on monitors[2].

A condition variable can help solve problems that could be complicated to solve using semaphores[1, 5]. It is supported by both POSIX and SDL[13]. A condition variable queue can only be accessed with two methods associated with its queue, typically called wait and signal. (In Java, they are called await and signal.) Threads wait for a guard [3] statement to become true to enter the queue and threads that change the guard from false to true could wake up the waiting threads. In practice, it always works with a mutual exclusion variable (mutex).

In our application, we use condition variables to declare a class whose objects are shared buffers. The following code segment shows how we can create shared buffer objects (buffer 1 and buffer 2 in Figure 2) by making use of SDL condition variables to define the class SharedBuffer:

```cpp
class SharedBuffer {
private:
   int head;
   int tail;
   SDL_mutex *mutex;
   SDL_cond *condVar; // condition variable
   int N; // number of buffer slots
   bool quit; // number of buffer slots
public:
   int frameSize;
   // Pointer to shared buffer
};
```
Frame *bufs;
// constructor, n0 buffer slots
SharedBuffer( int n0, int frameSize );
// get a frame from the buffer at head
void get ( Frame &f );
// insert a frame into buffer at tail
void insert ( const Frame &f );
// wake up all sleeping threads
void wakeupAll();
// check if buffer (bufs) is empty
bool empty();
//someone has to signal everyone to quit
// when there is no more data
void setQuit();
bool toQuit();
~SharedBuffer();
};

The variables head and tail point to the head and the tail of the circular buffer respectively. Each buffer slot can hold a Frame object. A frame is inserted at the tail and read at the head. The variable mutex, short for mutual exclusion, ensures that only one thread is allowed to access a critical section of the code. The variable condVar is a condition variable that works along with mutex to guard the execution of certain code section. The pointer variable bufs will point to the buffer slots that can hold a maximum of N frames. The memory for the buffer slots is allocated at the constructor, which can be implemented like the following:

SharedBuffer::SharedBuffer(int n0, int fSize)
{
    N = n0;  //number of slots in bufs
    frameSize = fSize;
    bufs = new Frame[N];  //allocate memory
    for ( int i = 0; i < N; i++ )
        bufs[i] = Frame ( frameSize );
    head = tail = 0;
    mutex = SDL_CreateMutex ();
    condVar = SDL_CreateCond();
    if ( mutex == NULL ) {
        cout << "Creating mutex failed!"; return;
    }
    if ( condVar == NULL ) {
        cout<<"Create cond variable failed!"; return;
    }
    quit = false;
}

Note that the statement bufs = new Frame[N]; only allocates memory to hold N Frame structures. It does not allocate memory to hold the video data. The allocation of memory for holding video data is done in the constructor of the Frame class in the for-loop.

The member function get of SharedBuffer for reading data can be implemented in a way similar to the following code:

doctor SharedBuffer::get( Frame &f )
{
    // acquire lock to lock
    // subsequent code section
    SDL_LockMutex ( mutex );
    /*
    if condition not met,release lock mutex
    and go to sleep; when woken, it has to
    reacquire lock before proceeding.
    */
    while ( head == tail && !quit )
        SDL_CondWait ( condVar, mutex );
    if ( head == tail ) { //no more data
        SDL_CondSignal ( condVar );
        SDL_UnlockMutex ( mutex );
        return;
    }
    int h = head % N;  //cicular buffer
    memcpy(f.buf,bufs[h].buf,frameSize);
    SDL_LockMutex ( mutex );
    head++;
    SDL_CondSignal ( condVar );
    SDL_UnlockMutex ( mutex );
}

A consumer thread uses the get function to read a frame at the head of the shared buffer. As the variables head and tail are accessed by multi-threads, before examining their values, the function has to acquire the lock mutex (by the statement SDL_LockMutex ( mutex );). If the lock has been acquired by another thread, the thread is blocked and waits until the other thread releases the lock and wakes up the waiting thread. While there is still data (not quit), the thread waits on the condition head == tail, which implies that the shared buffer is empty. The command SDL_CondWait() sends the thread to sleep if the guard is not true and releases the lock mutex. When it is woken by the other thread, it will try to acquire the lock mutex and check the guard again. In our case, as long as the buffer is empty, the thread releases the lock mutex and goes to sleep. The condition can be changed by a producer thread, which inserts data into the buffer, increments the tail value, and wakes up the sleeping thread. After it has been woken, the consumer thread tries to reacquire the lock mutex and checks the condition in the while loop again; if successful, it will proceed to execute subsequent code.

The C/C++ function memcpy, which is the fastest library routine for memory-to-memory copy in large blocks, is used to copy the video data from the shared buffer slot to the buffer of the referenced Frame object f. The Frame object is returned to the calling procedure as a pass-in-reference variable. Since the copy operation from the specified buffer slot will not affect other operations, and the head pointer has not advanced, it is not necessary to lock the op-
operation, meaning that other threads can proceed to perform other operations while data are copied. However, when we increment the head value, we must lock the operation as its value affects other operations.

The member function `insert` is implemented in a similar way:

```cpp
void SharedBuffer::insert ( const Frame &f )
{
    SDL_LockMutex ( mutex );
    while ( tail - head >= N )
        SDL_CondWait ( condVar, mutex );
    int t = tail % N;
    SDL_UnlockMutex ( mutex );
    memcpy ( bufs[t].buf,f.buf,frameSize);
    SDL_LockMutex ( mutex );
    tail++;
    SDL_CondSignal ( condVar );
    SDL_UnlockMutex ( mutex );
}
```

This code is similar to that of `get` except that now video data are copied from the passed-in frame buffer to the shared buffer slot. Again, we do not need to lock the `memcpy` operation even though it is writing data to the shared buffer. This is because a `get` operation is not allowed when `head == tail` and we have not advanced the tail pointer. This means that no thread will read the location while a thread is writing to it. On the other hand, we need to lock the operation when we increment the tail variable.

### 4. Producer-Consumer Implementation

With the `SharedBuffer` class defined, we can implement the producer-consumer paradigm of Figure 2 by instantiating two `SharedBuffer` objects, corresponding to `Buffer 1` and `Buffer 2` of Figure 2. Each of these buffers will be shared between the producer `dataSource` and the consumer `encoder` or `player`.

```cpp
SharedBuffer *sharedp1;
SharedBuffer *sharedp2;
......
int main( int argc, char *argv[] )
{
    ..... 
    sharedp1 = new SharedBuffer
                (nSlots, frameSize);
    sharedp2 = new SharedBuffer
                (nSlots, frameSize);
    ..... 
}
```

The producer `dataSource` inserts data into the shared buffers:

```cpp
int dataSource ( void *data )
{
    ..... 
    Frame aFrame (sharedp1->frameSize);
    while ( !sharedp1->toQuit() )
    {
        int nBytes = aFrame.readFrame(avi);
        ..... 
        sharedp1->insert ( aFrame );
        sharedp2->insert ( aFrame );
    }
    return 0;
}
```

The consumer `encoder` fetches a frame from the buffer and encode it:

```cpp
int encoder ( void *data )
{
    ..... 
    Frame aFrame ( sharedp1->frameSize );
    while ( !sharedp1->toQuit() ||
            !sharedp1->empty() )
    {
        sharedp1->get ( aFrame );
        encode_one_frame ( aFrame );
    }
    return 0;
}
```

Consumer `player` works in a similar way except that it fetches data from the second buffer:

```cpp
int player ( void *scr )
{
    ..... 
    Frame aFrame (sharedp2->frameSize);
    while ( !sharedp2->toQuit() ||
            !sharedp2->empty() )
    {
        sharedp2->get ( aFrame );
        display_frame ( aFrame );
    }
    return 0;
}
```

### 5. Results and Discussions

We tested the application by running it using a 64-bit Linux machine, which has eight CPU’s. We compared it with a single-threaded encoder, which uses the same encoding techniques. We find that the multi-threaded encoder can encode video data with a speed up factor up to 2 as compared to the single-threaded encoder. Moreover, we find that there was no memory leaking in the system even after it has run for a long time. Figure 3 shows a sample of compression; the left is the original image and the right is...
the restored image from the compressed data. The compression ratio in this sample is about 10, which is similar to that obtained by an MPEG-4 encoder as the compression techniques we have used are based on the MPEG-4 standard. By decoupling the tasks, the computation can be delegated to different machines, for example, one task running in a desktop machine and another task running in a mobile device such as an android phone[10, 11, 12, 13, 15].

6. Conclusion

In conclusion, we have presented a framework of implementing video data compression in C/C++, which exhibits robustness, efficiency, and flexibilities. It makes use of the producer-consumer paradigm to decouple tasks; each task is run in its own thread independently with a producer thread inserting data into a circular queue that a consumer thread extracts. Condition variables are used to ensure that certain conditions are required to meet before an operation of writing or reading data to or from a shared circular queue. A thread is blocked and goes to sleep when the conditions is
not met and is woken by an active thread, which changes the condition.

References