Multi-CPU Video Processing

Igor Jánoš*

Faculty of Informatics and Information Technology
Slovak University of Technology
Bratislava / Slovakia

Abstract

Graphic processors have developed significantly over the last few years allowing programmers to utilize their astonishing power for general purpose computation leaving classic CPUs behind. Much higher computational complexity, large picture dimensions and memory requirements of high definition video have made pure software real-time video processing impossible. Latest trend in CPU development is to provide more CPU cores in one chip providing higher performance for parallel applications. In this paper we would like to find out how thread-parallel video processing performed on multi-core CPUs can be used to accelerate processing of high definition video.

Keywords: High definition, Parallel, Video Processing, Thread, Dual-core CPU

1 Introduction

Video processing is a very important phenomenon nowadays. Many processing methods are widely used either in television systems, video postproduction or even in common life. Despite the fact that professional hardware video processing solutions exist, software video processing is very popular mainly because of the great flexibility it offers.

In the past few years there has been a tremendous development in powerful graphic processors (GPU) that have made very difficult computations possible. Statistical results show that GPU development nearly doubles the Moore's law by doubling the performance every six months. Now even mid-class GPUs can easily outperform latest CPU processors in various applications such as vector algebra, physics or particle systems.

There has also been a quite dramatic development in classic CPU processors during the last year and two major CPU vendors have entered the multi-core race announcing dual-, quad- and even eight-core CPUs. These multi-core CPUs offer a thread-level parallelism that may bring additional performance gain if used properly by software.

In this paper we would like to explore how thread-level parallelism may accelerate video processing algorithms on a multi-core CPU.

* janos@creeo.net

1.1 Video sequence and common issues

Basically a video sequence is a set of successive pictures changing in time. The inertia of human visual system makes pictures shown at discrete time intervals appear like a fluid video sequence. The minimal required frame rate for a video sequence to look fluid is somewhere around 25 frames per second. In the era of analog television interlacing was introduced as a sort of compression effectively doubling the frame rate while preserving the bandwidth necessary for transmission. In an interlaced frame the odd and even lines represent different instances of pictures in time thus their vertical resolution is reduced. Even though interlacing was invented in analog times it is still popular and widely used in modern compression methods that offer both interlaced and progressive coding. In order to convert an interlaced coded video into progressive special deinterlace algorithms must be performed to reduce the comb effect in high-motion areas. Figure 1 shows an interlaced frame processed by a motion blur deinterlace algorithm.

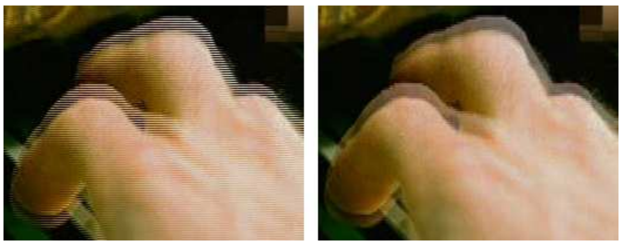


Figure 1: Interlaced (left) and motion blur deinterlaced (right) picture

Another example of a common video issue is camera noise. Noise usually tends to follow something like a normal distribution both in time and space. Many noise reduction methods have been proposed taking advantage of both spatial and temporal noise properties.

Many existing compression methods use block based motion compensation to reduce temporal redundancy in encoded video. Depending on the quality parameter visible discontinuities may appear on the edges of motion blocks. In the latest H.264 codec an in-loop deblocking filter is a mandatory part of the encoding/decoding process [2] but most of the existing codecs do not possess such a feature. Therefore deblocking is a very useful quality increasing post-processing method.

1.2 High definition video

In order to process high definition video much higher computational power is required as well as higher memory transfer capability. High definition frame with dimensions of 1920 x 1080 pixels in planar YUV 4:2:0 format needs 4 MB of memory. When compared to a standard resolution frame with dimensions of 720 x 576 the amount of memory to keep and process is five times larger.
The structure of this paper is following. In section 2 we present research that has been done in the field of parallel software-only video processing. Section 3 provides an overview of algorithms used in this paper and describes the logic of proposed parallel processing algorithm. In section 4 we summarize results and display them in table and graph. Section 5 gives us some hints what the next subject of research may be.

2 Previous work

The idea of dividing a larger task into several smaller steps that can be executed in parallel is not new. Many studies have been conducted on the topic of the video processing.
A Parallel Software-Only Video Effects Processing System [8] (PVPS) project adopts the parallel approach to the video processing and demonstrates how networked workstations can replace a professional and expensive video production switcher. The PVPS represents a distributed approach that has been very popular in the past few decades where all subtasks have been performed on separate standalone nodes. Each node used in the PVPS was a standalone PC with its own resources. Nodes were connected using a limited-bandwidth network. Since the nodes had not shared a common memory, lossless compression methods had to be introduced to reduce the number of bits transmitted over network.

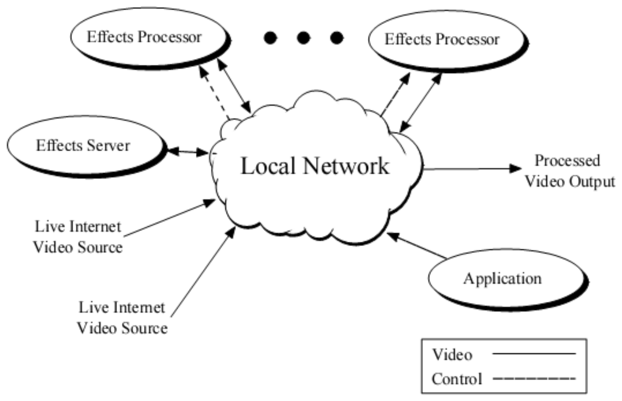


Figure 2: PVPS system overview

Main PVPS entity (*Application*) requires video processing by passing frames to *Effects Server* which is responsible for subsequent *Effects Processors*' calls. Finally a resulting picture is again encoded and sent to output. The PVPS system was capable of processing QVGA video using a 10Mbps LAN.

Later on a more advanced system [9] Gigabit Ethernet-based Video Processing system (VP) was designed on the basis of the PVPS to process the full D-1 resolution video with the dimensions of 720 x 576 pixels using a gigabit Ethernet to connect the processing nodes. Among other things the VP project has confirmed that the network bandwidth requirements have risen significantly with the increase of video dimensions and the number of processing nodes and that more efficient compression method for transferred data had to be used.
In general parallel and distributed computations on a larger scale using networked clusters of independent computers provide significantly larger performance gain when compared to single CPU system [7] however the transfer speed may turn out to be a big problem when processing images in very high resolution.
Architecture of a PC with a multi-core CPU offers analogous means of parallel processing. Threads can take place of independent processor entities and the system bus makes the system memory accessible to the CPUs. When processing high definition video the memory transfer requirements are at such a high level that memory performance is probably the most crucial factor in overall video processing performance.
Another form of parallelism that may be utilized to maximize performance is data parallelism at instruction level [6]. SIMD instruction sets are integral part of modern CPUs and are widely used nowadays.

3 Performed algorithms

This section provides an overview of implemented algorithms used in this paper as well as IDCT experiment results using multiple threads. Later in this section the details of a proposed parallel algorithm are explained.

3.1 Performance gain on IDCT

Inverse discrete cosine transform is a typical example of an algorithm with low memory transfer and high computational power demands. A study [1] from 2005 has proposed several methods to perform an inverse discrete cosine transform using a generic GPU that were able to outperform an MMX optimized CPU algorithm but were significantly slower than SSE optimized IDCT.
We have written a simple application that performs IDCT on an 8x8 matrix of 16-bit integers in one, two and four threads on several processors using either MMX or SSE instruction sets if available.
The graph displayed in figure 3 shows nearly a 2x speedup for dual-core CPUs.

```
void idct_thread(void *param)
{
    int16 block[64];
    while (GetTickCount() < stoptime) {
        for (int j=0; j<1000; j++)
            idct_8x8(block);
        InterlockedIncrement(&count);
    }
}
```

Figure 2: IDCT thread code snippet

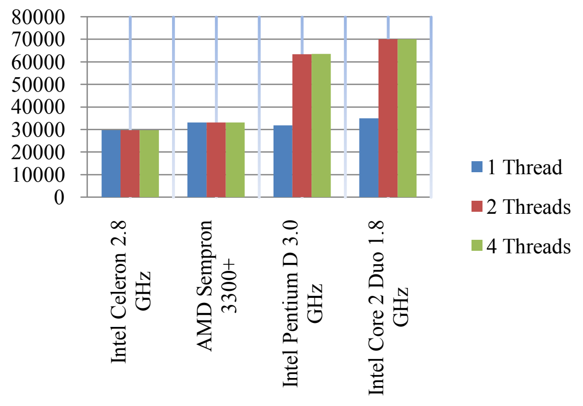


Figure 3: IDCT performance for several tested CPUs

An Intel corporation microprocessor research labs study [4] indicates that there is some space left for optimizing for processors with Hyper-Threading technology that can produce a speedup up to 17% for IDCT calculation.

3.2 Gradual denoise filter

The gradual noise reduction filter was introduced in a DScaler project [5]. We have decided to use it in this paper because it offers good visual quality and can be performed in separate threads.
This filter calculates the sum of absolute differences between a four pixel horizontal block in the current frame and the same block in the preceding frame. This difference measure is used to determine the kind of averaging which will be conducted. If it is more than the *noise reduction parameter*, motion is inferred. In that case new pixel values are used. If it is less than the *noise reduction parameter*, we use the ratio of *difference / noise reduction* to determine the weighting of the old and new values.

$$N = \sum_{i=0}^{3} |oldPixel_i - newPixel_i| \qquad (1)$$

$$R = Noise\ Reduction\ Parameter$$

$$M = \begin{cases} 1; \frac{N}{R} \geq 1.2 \\ 0.999; 1.2 > \frac{N}{R} \geq 1 \\ \frac{N}{R}; otherwise \end{cases} \qquad (2)$$

$$Result\ Pixel = oldPixel \times (1 - M) + newPixel \times M \qquad (3)$$

Other commonly used noise reduction filters such as mean filter or median filter reduce spatial noise and may reduce picture details much more than this gradual algorithm. Gradual denoise algorithm can also be efficiently computed using SSE extension set and threads.

3.2 Color controls filter

The second applied filter algorithm modifies brightness, contrast and saturation of an input picture according to the following equations:

$$Y' = 128 + \left(\frac{contrast \times (Y - 128)}{128}\right) + brightness \qquad (4)$$

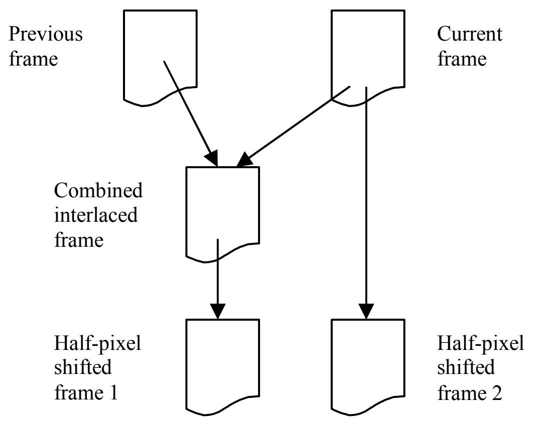
$$CR' = 128 + \left(\frac{saturation \times (CR - 128)}{128}\right) \qquad (5)$$

$$CB' = 128 + \left(\frac{saturation \times (CB - 128)}{128}\right) \qquad (6)$$

All values are clipped within range of 0..255. It operates on one frame only so the memory requirements are lower when compared to the gradual denoise filter that makes use of two frames. The color controls filter can also be computed in parallel threads because there are no dependencies among resulting pixel values.

3.2 Deinterlace and double rate filter

The last algorithm used in this study is a combined deinterlace and double frame rate filter. The purpose of this algorithm is to generate a frame to represent a video scene at time between two successive frames. A very cheap way without using any expensive motion estimation is to combine odd lines from the previous frame with even lines from the current frame. This will introduce a comb effect such as in interlaced coding.
Finally both frames (the current frame and generated one) are shifted in vertical direction by half pixel to smooth out the comb effect. Pixel values in the destination frame are computed as an average between odd and even lines from the source frame.
A real algorithm implementation performs both phases at the same time. When creating the first output frame it uses the previous frame as a source for odd lines and the current frame as a source for even lines. When creating the second output frame it takes only the current frame as reference.



Previous frame
Current frame
Combined interlaced frame
Half-pixel shifted frame 1
Half-pixel shifted frame 2

Figure 4: Deinterlace and double rate filter

This algorithm can be implemented very efficiently using the SSE instruction set because of the `pavgb` instruction. Its performance depends mostly on the memory transfer speed because it needs to access three video frames at the same time.

3.3 Algorithm design

Our proposed approach deals with two worker threads for each processing filter and two additional threads – the main decoding thread and the master processing thread. The actual number of worker threads may be larger when running on quad-core CPU. Since we only had a dual-core CPU and the IDCT experiment did not show any reasonable performance gain with four threads we have set the number to two. Two synchronization events exist for each of the worker threads. One event to signal that a new command for the thread has been issued and a second event to signal that the worker thread has finished its job and is now idle and waiting for another command.
The job of the master processing thread is to set proper source and destination buffers, set the number of lines to process including starting offset and to activate the worker threads. When all worker threads of a processing filter signal completion, the next processing filter is activated.

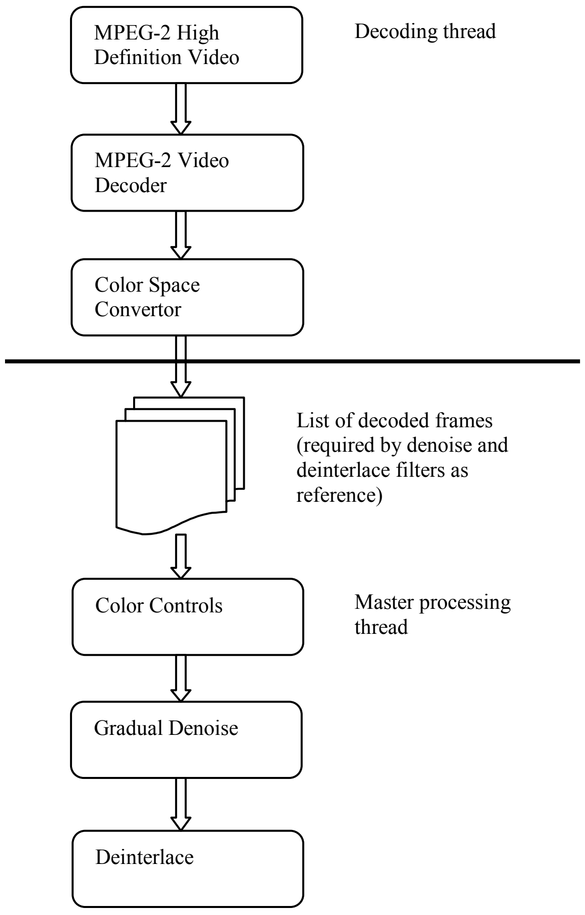


Figure 5: Sequence of operations in the experiment

To measure the algorithm performance we have set up a following experiment illustrated on figure 3. On a dual-core 3.2 GHz Intel Pentium D CPU we wanted to decode a 25 Hz high definition MPEG-2 video sequence, adjust the contrast, apply the gradual denoise filter and deinterlace using the mentioned deinterlace filter that produces output frames at a rate of 50 Hz.
Let us consider the video sequence and decoding module a black box capable of decoding high definition video frames at 75 frames per second (pure decoding without any display routines) and utilizing only one CPU core.
The main application is split into two main threads – a decoding thread and a master processing thread. As soon as the decoding thread has a frame decompressed it synchronizes with the master processing thread and forwards the decompressed frame for processing.
All implemented processing filters operate in packed YUV 4:2:2 color space so a color space conversion from the internal decoder format is performed before the sample is actually forwarded for processing. Packed YUV 4:2:2 (also known as YUY2) color space can directly be displayed using a hardware overlay surface so any additional color space conversion is not necessary for display routines. The decoding thread immediately continues with the decoding of another frame.
The master processing thread controls the processing filter worker threads and once all workers signal completion it waits for further instructions from the decoding thread. The gradual denoise filter operates on one input frame so that each of the two workers process one half of the frame. The same is true for the color controls filter. The deinterlace worker threads produce one full frame each.

4 Results

We have performed several test runs with 2 video sequences. The first one was a high definition trailer encoded at a data rate of 14 Mbits/s and the second one was a terrestrial DVB dump of a 4 Mbit/s standard definition TV show.

	SD (720x576)	HD (1920x1080)
Decoding only (no color conversion)	304.0 FPS	74.8 FPS
Decoding with color conversion	262.8 FPS	61.4 FPS
Full processing in 1 thread (original)	98.4 FPS	17.57 FPS
Full processing multi-threaded	142.6 FPS	25.87 FPS
Performance gain	44.9 %	47.7 %

Table 1: Processing performance results

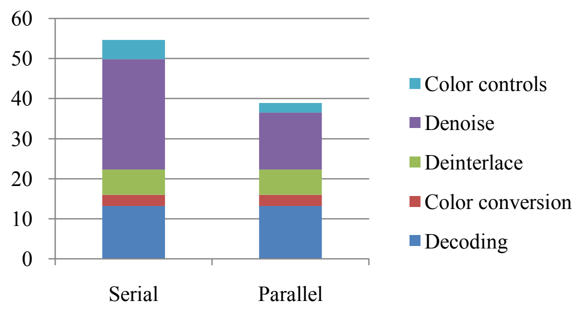


Figure 6: HD - time per operation in milliseconds

As seen from table 1 the application was able to decode and process high definition video in real time using only means provided by the local CPU. This was not possible with decoding and processing executed on one CPU core. Results also show a very similar performance gain for the standard resolution video sequence as well.
Figure 6 shows that processing time for color controls and denoise filters were successfully cut nearly at half while deinterlace time remained the same due to memory transfer limits.
Experiments with twice the number of processing worker threads did not show any reasonable performance gain on the tested processors (Intel Pentium D @ 3.0 GHz, Intel Core 2 Duo @ 1.8 GHz).
It is also worth mentioning that due to the serial nature of the decoding module and the decoding-processing thread synchronization the total CPU utilization was only about 75% which leaves enough space for additional operations that may need to be performed such as container file parsing and audio stream decoding.

5 Conclusion and future work

This experiment has proven that thread-parallelism can make software real-time high definition video processing possible. With quad- and eight-core CPUs to come the only bottleneck appears to be the memory performance. Perhaps a CPU cache large enough to fit several high definition frames may help to reduce the memory access time penalty.
Several studies have shown that a generic GPU can be used to accelerate several parts of a video decoding process such as the color space conversion and the motion compensation. It could be interesting to explore a possibility of implementing a hybrid video decoder that would utilize both the thread-parallelism and the GPU acceleration to speed up the video decoding/processing operation.

6 Acknowledgements

I wish to thank to Martin Šperka for his suggestions on this paper and to Peter Janičkovič for giving me access to several PC systems to perform experiments on.

7 References

[1] Bo Fang, Guobin Shen, Shipeng Li, Huifang Chen. *Techniques for Efficient DCT/IDCT Implementation on Generic GPU.* ISCAS 2005

[2] Iain E.G. Richardson. *H.264 and MPEG-4 Video Compression.* John Wiley & Sons, England, 2004

[3] Han Chen, Kai Li, Bin Wei. *Memory Performance Optimizations for Real-Time Software HDTV Decoding.* IEEE ICME 2002, August 2002

[4] Yen-Kuang Chan, Eric Debes, Rainer Lienhart, Mathew Holliman, Minerva Yeung. *Evaluating and Improving Performance of Multimedia Applications on Simultaneous Multi-Threading.* International Conference on Parallel and Distributed Systems, Taiwan, 2002

[5] http://deinterlace.sourceforge.net : DScaler project

[6] G. Conte, S. Tommesani, F. Zanichelli. *The Long and Winding Road to High-Performance Image Processing with MMX/SSE.* Fifth IEEE International Workshop on Computer Architecture, 2000

[7] X. L. Li, B. Veeravalli, C. C. Ko. *Distributed Image Processing on a Network of Workstations.* International Journal of Computers and Applications, 2003

[8] K. D. Mayer-Patel. *A Parallel Software-Only Video Processing System.* Dissertation project, 1999

[9] H. Eidenberger. *Gigabit Ethernet-based Parallel Video Processing.* Proceedings of the 11th International Multimedia Modelling Conference (MMM05), 2005