SCENE BOUNDARY DETECTION TECHNIQUE BASED ON BOTTOM-UP ATTENTION SYSTEM AND OPENCL PARALLEL IMPLEMENTATION

S. Axyonov, D. Kovalenko, I. Potapyev

National Research Tomsk Polytechnic University

E-mail: axoenowsw@tpu.ru, wf34@sibmail.com, ipotapev@gmail.com

This paper spotlights the maintaining of scene boundary detection system in video and process of porting it to the OpenCL. The scene boundary detection algorithm proposed by authors is based on bottom-up focus attention principle. The system builds Gaussian pyramids from input image, calculates map of saliency from the image and then detects the most salient regions. The scene cut is detected when correlation of previous and next image's histograms of salient areas is lower than threshold. To decrease the algorithm's running time a parallel architecture was implemented. Several tricks and techniques which served for reaching better performance are described.

Keywords: scene detection, pyramid of image, video analysis, OpenCL, parallel processing.

Introduction

In order to process video data efficiently, a video segmentation technique through scene cut detection must be required. This is a fundamental operation used in many digital video applications such as digital libraries, video on demand (VOD), etc. There is set of approaches used for solving this task. In [1][2] authors propose system for automatic detection of replay segment based on image templates. Paper [3] describes algorithm of processing a compressed digital video MPEG bitstream to detect scene changes. The main feature of chosen algorithm is ease of paralleling code development to increase the performance. In this paper, we proposed an approach based on using bottom-up attention[4].

The proposed algorithm contains huge amount of consecutive operations, where result of the previous one doesn't affect result of the next one. It means that such kind of applications contain a huge amount of the parallelism. Hence, execute it in a serial manner would be ineffective and slow. Our goal was to take advantage of the natural parallelism of the task and implement the parallel version of the system.

Hence, we had a goal to pick parallel programming technology which is suitable for servers and desktops. Most of them are highly specialized. For example, MPI oriented for execution on clusters and OpenMP devoted for creating parallel code for SMP systems. Finally, OpenCL was chosen. This decision is supported with following reasons:

- General purpose GPU is the approach acceptable for desktops and servers both
- OpenCL is oriented to solving mathematical tasks
- OpenCL technology supported with all vendors (NVIDIA, ATI, Intel , etc.)

Algorithm

The system takes video file as input by using OpenCV functions for consequent access to the frames. Each frame of video is represented as RGB image. Next, Gaussian pyramids of the frame are created for construction maps of characteristic features. Gaussian pyramid is a set of images that are duplicates of each other and where the next image is smaller than the previous one in 2 times[5]. The number of levels of the pyramid is determined by the minimum allowable size of the image within the pyramid. The Gaussian blur is used to avoid the negative impact of pixelation when building pyramid.

The system builds five pyramids for each frame. The first pyramid characterizes the intensity of the image.The other four pyramids consist respectively of the four images, each of which represents a specific color channel, red, green, blue, or yellow, negative values are set to zero [6].

First set of feature maps is related to intensity contrast of image and consist of six maps I(c,s). Each of them is calculated by formula

$$I(c,s) = |I(c) - I(s)|$$

Where $c \in \{2,3,4\}$ and $s = c + \sigma$, $\sigma \in \{3,4\}$ are index levels of pyramid, «-» is center-surround difference[7].

Next sets of feature maps are connected with color input. Two sets of feature maps are combined by contrast principle from R, G, B, Y pyramids:

$$RG(c,s) = |(R(c) - G(c)) \ominus (G(s) - R(s))|$$
$$BY(c,s) = |(B(c) - Y(c)) \ominus (Y(s) - B(s))|$$

Feature maps are combined into three "conspicuity maps", I_{cons} for intensity, C_{cons} for color at the scale ($\sigma = 4$). They are calculated by across-scale addition, $\oplus$, which consists of reduction of each map to scale 4 and point-by-point addition:

$$I_{cons} = \oplus_{c=2}^{4} \oplus_{s=c+3}^{c+4} I(c,s)$$
$$C_{cons} = \oplus_{c=2}^{4} \oplus_{s=c+3}^{c+4} [RG(c,s) + BY(c,s)]$$

Next, the two conspicuity maps are normalized and summed into the final input S to the saliency map which is used by focus attention subsystem for the computation of the mask frame[8]. The motivation for creation of two separate channels, $\bar{I}$ and $\bar{C}$, is the hypothesis that similar features compete strongly for saliency, while different modalities contribute independently to the saliency map. Saliency map is calculated by naïve summation[9].

According to figure 1, the location of the areas with the greatest intensity on feature map significantly correlated with the location of the main objects on the frame. Thus, finding the local maximum on the map and use of adaptive threshold cutting off the area around the local maximum allow to apply these areas on the mask for obtaining frame information which is important for the efficient detection of scene boundaries.

Obtained salient areas define the regions on the image where histogram correlation would be computed for previous and next frames

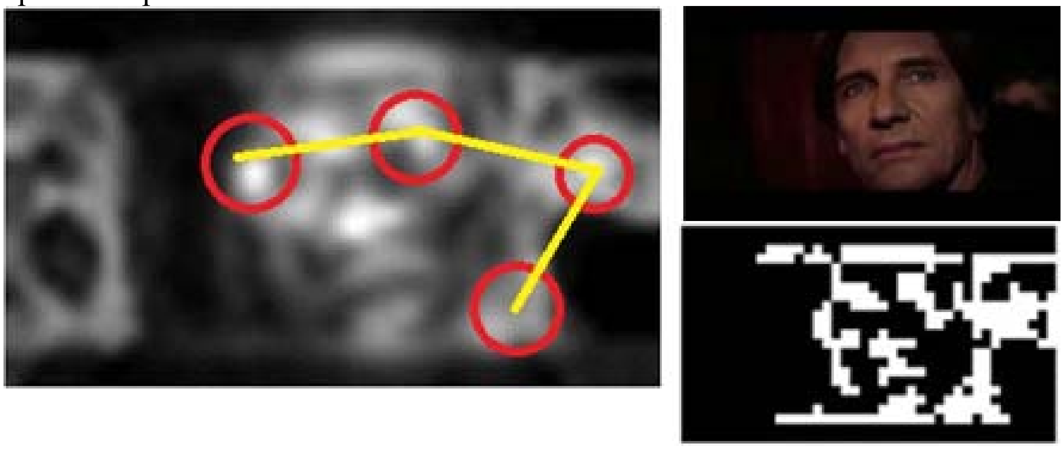

Figure 1. Salient regions on an image

The correlation function is described by the formula:

$$d(H_1,H_2)=\frac{\sum_I (H_1(I)-\bar{H}_1)(H_2(I)-\bar{H}_2)}{\sqrt{\sum_I (H_1(I)-\bar{H}_1)^2 \sum_I (H_2(I)-\bar{H}_2)^2}},$$

where H_1 – histogram of the previous frame, H_2 – histogram of the current frame $\bar{H}_k=\frac{1}{N}\sum_J H_k(J)$, N – number of histogram bins.

Thus, when passing of frames is done, the frame correlation vector for all frames is built. Frames where correlation is close to 1.0 are frames of one scene. The low values of the correlation with a high probability indicate a scene cut between these frames. Filtering of the vector on the threshold gives a number of pairs of frames located on the boundaries of scenes. Filtration is performed by using of adaptive threshold:

$$T=\sum_{i=0}^{2n} C_{ki}\varphi_i\ , \qquad \varphi(C)=\frac{1}{\sigma_c\sqrt{2\pi}}e^{-(c-M_c)^2/2\sigma_c^2}\ .$$

Where T – threshold value, φ_i – weighting factor, n – number of neighbors of the analyzed element, M_c – mean value, σ_c^2 – variance, $C_n=\frac{d(R_{N-1},R_N)+d(G_{N-1},G_N)+d(B_{N-1},B_N)}{3}$ – elements of the analyzed vector, $d(x,y)$ – correlation function, R_i,G_i,B_i – vectors obtained from RGB histogram of the current frame. Obtained vector contains binary values, where 1 indicates a cut between scenes.

Architecture of parallel solution

Architecture of the system was slightly reconsidered and rewritten with the purpose to develop clear and supportable parallel code. Writing of parallel code with use of any technology has lots of pitfalls and dramatically slower than writing serial code. So, rewriting of the architecture allowed to deliver several important features.

First feature is that OpenCL and C++ versions are interchangeable. They represented as classes PyraProbe and OCLPyraProbe on figure 2. These classes encapsulate all business-logic related to pyramidal analysis of the frame (all computationally expensive work). Figure 3 shows that at the highest level of application call stack code operates with IBasicPyraProbe objects without even knowing which implementation it is.

Also, both versions are kept in the same project in purpose of easier building and execution in both ways. The library is being built with or without OpenCL support based on preprocessor definitions, and being executed serial or parallel way based on given settings.

Another architecture feature is namespaces Processors and OCLProcessors which shown on figure 3. Both of them contain image processing functions which are exploited by PyraProbe and OCLPyraProbe respectively. Some examples of these functions are listed for better insight of the idea:

- generation of the 2D Gaussian distribution,
- 2D convolution,
- normalizing color image,
- creating image pyramid.

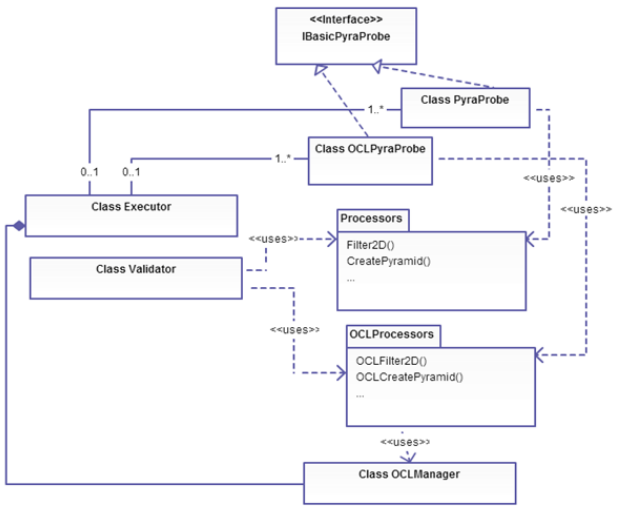

Figure 2. Architecture of the developed system

This design allowed to develop parallel version iteratively and enabled unit-testing of these functions. Class Validator serves for this functionality. So developers were able to prove that some function ported successfully by comparison of the results of C++ and OpenCL version of the function.

OCLManager serves for handling all SDK-related calls. It builds and stores kernels, keeps OpenCL context and other SDK objects required for execution.

Other aspects of the architecture:

- Minimized interaction between RAM and videomemory. Copying of image occurs only twice: delivering source image to OCLPyraProbe and then acquiring result back to RAM. All computation in between is being performed on GPU.
- Modularity of the system enables easy code adaptation for execution on the GPU farm.

When development of the system with architecture described above was finished, performance tests have been performed.

These tests revealed slight improvement at the high-resolution video, but parallel version didn't outperform serial one on the low-resolution videos.

Most of the application running time was taken by convolution calculation. Filtering is so such computationally expensive because model uses big kernels (25x25, 21x21). So, filter2D is the most promising target for optimizations. One of the properties of the efficient GPGPU code is low divergence. This means that all threads of kernel have approximately same execution time. If control flow of the kernel has a lot of conditional statements, most likely that code will be slow [10].

Removing of the conditionals gave significant speed gain for our case.

OpenCL global memory has huge capacity but the slowest access speed. So, the only way to implement something efficient is to put an effort on extracting advantage of the local and constant memory zones.

There is possibility to define memory buffers as constant if buffers are read only. Also size of the constant buffer is bounded by value which depends on computing device. So, first thing to do was to move all filter buffers from global memory to constant.

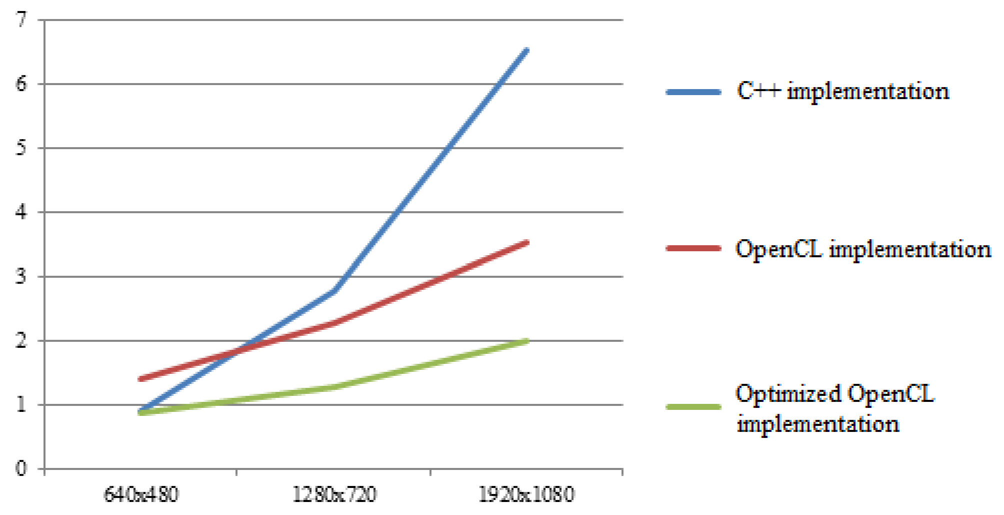

Figure 3. Time comparison of processing one frame

Another way to speed up convolution operation is to avoid reading neighbor pixel values from the global memory [11]. So, filter2D kernel execution model was changed to use workgroups of size 16x16. Items of the same workgroup load part of the image to the local memory and then use this data for convolution calculation.

It is good practices to keep amount of kernel arguments as small as possible. As it obvious from table 2, filter2D was separated to two kernels. This action enables us to not pass size of kernel (and some other variables like that) as argument, but define it as a constant. These values are used in the loop code; consequently, compiler will be able to implicitly unroll loops [12].

Our optimization work is not finished yet, but even now it is obvious that GPGPU solution achieved performance on this problem of image analysis which is way more effective than serial CPU solution. Tests which are represented in figure 3 ran at desktop with Intel i5-430m processor and AMD Radeon HD 5850M.

So, plans for future development are:

- Continuation of the OpenCL code optimization
- Work on the computational model for increasing precision and decreasing computational complexity.

This research was partially supported by the “Nauka” program of Ministry of Education of Russian Federation, contract number 8.8113.2013 and a grant from Russian Foundation for Basic Research RFFI 13-07-00397A.

References

1. Pan, Hao, Baoxin Li, Sezan, M.I. Automatic detection of replay segments in broadcast sports programs by detection of logos in scene transitions // Acoustics, Speech, and Signal Processing (ICASSP). 2002.

2. Papageorgiou C.P., Oren M., Poggio T. A General Framework for Object Detection // Proceedings of the Sixth International Conference on Computer Vision. 1998.

3. Jianhao Meng, Yujen Juan, Shih-Fu Chang. Scene change detection in an MPEG-compressed video sequence // Proc. SPIE 2419, Digital Video Compression: Algorithms and Technologies 1995. 14.

4. Kovalenko D., Potapyev I. Scene boundary localization based on contrast region analysis // Tomsk Polytechnic University, MSIT №10. 2012. – P. 60–62.

5. Burt P.J., Adelson E.H. The Laplacian Pyramid as a Compact Image Code // J. IEEE Transactions on Communications Communications. Volume 31, Issue 4. 1983. 532–540.

6. Ware C. Color sequences for univariate maps: theory, experiments and principles // Computer Graphics and Applications, IEEE. Volume 8. Issue 5. 1988. 41–49.

7. Sun-Gu Sun, Dong-Min Kwak, Won Bum Jang, Do-Jong Kim. Small target detection using center-surround difference with locally adaptive threshold // Image and Signal Processing and Analysis. Proceedings of the 4th International Symposium on. 2005. 402–407.

8. Zhaoping Li. A saliency map in primary visual cortex // Trends in Cognitive Science. Volume 6. Issue 1. 2002. 9–16.

9. Xiaodi Hou, Liqing Zhang. Saliency Detection: A Spectral Residual Approach // Computer Vision and Pattern Recognition, IEEE Conference on. 2007. 1–8.

10. In Kyu Park, Singhal N., Man Hee Lee, Sungdae Cho. Design and Performance Evaluation of Image Processing Algorithms on GPUs // Parallel and Distributed Systems, IEEE Transactions on. 2011. Volume 22. Issue 1. 91–104.

11. Goorts Patrik, Rogmans Sammy, Vanden Eynde Steven, Bekaert P. Practical examples of GPU computing optimization principles // Signal Processing and Multimedia Applications (SIGMAP), Proceedings of the 2010 International Conference on. 2010. 46–49.

12. Han Dong, Ghosh, D., Zafar, F., Shujia Zhou. Cross-Platform OpenCL Code and Performance Portability Investigated with a Climate and Weather Physics Model // Parallel Processing Workshops (ICPPW), 2012 41st International Conference on. 2012. 126 – 134.