International Journal of Advancements in Electronics and Electrical Engineering– IJAEEE
Volume 4 : Issue 2 [ISSN : 2319-7498]
Publication Date : 30 October, 2015

Accurate and Efficient Visual Search on Embedded Systems

[M. Balestri, G. Cabodi, G. Francini, A. Garbo, C. Loiacono, D. Patti, S. Quer]

***Abstract*— Recent developments in embedded processors have enabled heterogeneous computing on mobile devices using open-access general-purpose computing languages. Following the MPEG CDVS standard, this paper presents an efficient feature computation phase, completely implemented on embedded devices supporting the OpenCL framework. Following our contribution to the MPEG-CDVS standard, we present the new born CDVS detector and its design for multi-core parallel GPUs. We show how to adjust algorithmic choices and implementation details to target the intrinsic characteristics of the embedded platforms selected. We compare our GPU implementation of the ALP keypoint detector with the CPU based implementation of the CDVS standard. We present data on different GPUs showing that our solution is up to 7x faster than the CPU version. To sum up, one of the main feature of our algorithm is to be fast enough to be able to open new visual search scenarios exploiting entire real-time on-board computations with no data transfer.**

***Keywords*— image processing, visual search, feature detection, compact descriptor, general purpose GPUs, embedded systems**

I. Introduction

Smart camera phones and tablet PCs have shown great potential in mobile visual search, thanks to the integrated functionality of high resolution color camera, powerful SoC and pervasive 3G wireless connection. Most of the existing mobile visual search systems are deployed in client-server architectures. In on-line applications, the client takes the picture and transfer it to the server. The server maintains a duplicate or near-duplicate visual search system, and employs approximate visual matching techniques. In other words is the server that identifies the most similar images, and returns visual objects information to mobile users.

Unfortunately, in 3G wireless environments, the upstream delivery of a visual query is subject to the network constraint of unstable or limited bandwidth. Latency from query delivery may degenerate the user experience in a significant way. However, with fast growing processing power in mobile devices using Graphical Processing Units (GPUs), sending an entire image seems unnecessary, since visual feature extraction and compression can be performed on mobile devices.

To reduce the visual query delivery latency, off-line applications need compact and discriminative visual descriptors [1]. This trend has received dedicated efforts in MPEG standardization, i.e., the Compact Descriptor for Visual Search (CDVS) [2]. The compact descriptor of an image is composed of two main elements, namely a selected number of compressed local descriptors and a single global descriptor, representing the whole image.

These two elements are produced by a series of processing steps:

- **Keypoint detection**: Identification of "Low Polynomial degree" (ALP) keypoints. This phase is based on the creation of a scale space made by a set of Laplacian of Gaussian filtered images and the subsequent identification of extreme by means of polynomial approximations.
- **Orientation Assignment**: In this step, one or more orientations are assigned to each keypoint, based on local image gradient directions. This is the key operation to represent keypoint descriptors as a function of their orientations, therefore achieving invariance to image rotation.
- **Feature selection**: Selection of a limited number of keypoints to maximize the quality for subsequent matching.
- **Local descriptor computation**: Computation of local descriptors in correspondence of the selected keypoints. Local descriptors are used in the pairwise matching phase, i.e., to find out whether the query and the reference images depict the same scenes or not. For match detection, geometric consistency check is performed to determine the number of inliers among the keypoint matches (correspondences) between the two images. In case of a match, localization information is provided, i.e., the position of the matching objects in the image, where homography estimation is conducted.
- **Local descriptor compression**: Scalar quantization-based compression of the selected local descriptors. CDVS supports different sizes of compact descriptor footprint, spanning from a maximum of 16KBytes per image, which is the fully performing operating mode, down to 512Bytes, for extremely constrained bandwidth scenarios.
- **Coordinate coding**: Compression of the coordinates of the selected keypoints.
- **Global descriptor aggregation**: Aggregation of local descriptors, to form a single global descriptor.

Gianpiero Cabodi, Alessandro Garbo, Carmelo Loiacono, Denis Patti, Stefano Quer
Politecnico di Torino
Italy

Massimo Balestri, Gianluca Francini
Telecom Italia
Italy

This research was supported by industrial contract 30/2014 entitled "Algorithms Optimization for Visual Search" .

A. Contributions and Comparisons

In this paper, we present an efficient and accurate OpenCL implementation of the first and the second steps listed above, as those steps are among the most critical ones. Our approach entails the following major contributions:

- Presenting the new ALP (A low degree polynomial) detector, the latest contribution to the CDVS standard.
- Re-engineering the Gaussian Scale Space (GSS) computation, i.e., the most expensive step of the keypoint detection phase.
- Optimizing all sequential steps of the CDVS standard to concurrently run on many-cores general purpose graphical processor units.
- Using new approaches to store all required data on proper GPU data structures to reduce the memory accesses and efficiently distribute the OpenCL kernels workload.

We present experimental results on standard benchmarks, comparing accuracy and efficiency of our GPU implementation with respect to its state of the art CPU counterpart.

Notice that, previous work (for examples, see references [3, 4, 5, 6]) present different implementation of the well known SIFT algorithm. Conversely, we concentrate on the novel CDVS ALP detector and on embedded platforms. Our target is to write an efficient and accurate ALP parallel implementation running on embedded systems. As far as we know, a few of those contributions are presented for the first time.

B. Roadmap

Section II and Section III describe our parallel implementation of the keypoint detector and orientation assignment algorithms, respectively. Section IV presents our experimental evidence. We run our experiments on standard images, and we present time performance as well as precision accuracy. Finally, Section V concludes the paper with some summarizing remarks. Notice that there is no explicit background section as background notions are reported in each section whenever required.

II. Parallel ALP Keypoint Detector

The ALP (A Low-degree Polynomial) [7] detector identifies interest points finding local extrema in the scale space by approximating the scale-space [8] using polynomials. In the scale space-representation, the images that result from the Laplacian-of-Gaussian filtering are functions of the scale parameter. ALP approximates these functions with polynomials of low degree. The algorithm works by subdividing the scale space in octaves in order to maintain a low complexity.

In the following sections (from Section II.A to Section II.F) we present the overall process for a single octave focusing on our GPU implementation of each step.

A. Gaussian Scale Space Computation

In order to compute Gaussian Scale Space (GSS) it is necessary to create a representation of the image frequencies. This is achieved using a scale space pyramid as introduced by Witkin [9]. The scale-space pyramid is constructed by taking a gray-scaled version of the original image and applying a Gaussian filtering repeatedly with kernels of increasing size. The most blurred image from this pyramid is down-scaled by a factor of two, and, afterwards, convolved with the same set of Gaussian kernels to create the next pyramid.

We implemented 2D Gaussian filtering using two 1D Gaussian horizontal filters. The process implies filtering the image with an 1D Gaussian, taking the transpose of the result, convolving (again) with the same filter, and transposing the result. We decided to use just horizontal filters due to the OpenCL data structure we consider. We used RGBA OpenCL textures in order to reduce the number of memory accesses. In fact, working with OpenCL textures drastically reduces memory access latency, whereas using OpenCL global memory is very time consuming. Moreover, most modern GPUs have a separate texture cache. When a specific pixel in the texture is requested, the GPU stores the data in a special buffer that is close to where the actual computations are performed. In our scenario, we work with gray scale images. As a consequence, we store 4 gray level pixels in one item of the RGBA texture, reducing the number of memory accesses. In this way we also reduce the original image width by 4 times as each OpenCL kernel[1] instance computes the convolution considering 4 gray level pixels at the same time. We also implemented a branch-less OpenCL code, as loops in OpenCL kernel functions are fully unrolled without branch ("if" statement) operations.

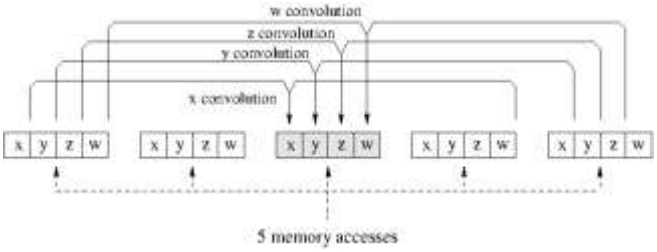

Fig. 1. Gaussian Filtering: Performing Convolution using OpenCL kernels.

Figure 1 shows a convolution step (with 4 gray pixels, a texture item), performed by a kernel. Boxes in the image represent different items of a texture. Each item is composed by 4 elements *(x, y, z, w)* representing 4 gray scale pixels. The size of the Gaussian filter is 15, then for each gray pixel the 14th adjacent pixels take part in the computation. In order to read the adjacent pixels, we need to perform just 5 memory accesses because, we read 4 gray pixels at the same time. The neighboring kernels can use this data to avoid further accesses to the texture memory. Furthermore, the kernels at the border of the image also apply padding, and

[1] OpenCL kernels can be seen as entry points to the device program, i.e., the only functions that can be called from the host. A single kernel execution can run on all or many processing elements in parallel.

consider padded pixels in the convolution computation. We didn't use OpenCL work-groups[2] to perform convolutions, because in this case the work-group setup is very expensive. We save about 10% of time avoiding work group usage in the convolution step. The Test Model (TM) implementation of the CDVS [2] performs the Gaussian Filtering using a 1D horizontal filter and a 1D vertical filter without transposing the result. Using a Gaussian filter of size 15, the standard approach implies 15 memory accesses, whereas our solution needs just 5 memory accesses. Even though we have to perform a matrix transposition, our solution has been proved to be much faster than the CDVS TM one.

B. *Laplacian Computation*

The Laplacian of an image highlights regions of rapid intensity change. The Laplacian operator is applied to an image that has first been smoothed by a Gaussian filter. The operator takes a single gray-level image as input and produces another gray-level image as output. We implemented OpenCL kernels able to apply the Laplacian operator to 4 gray pixels.

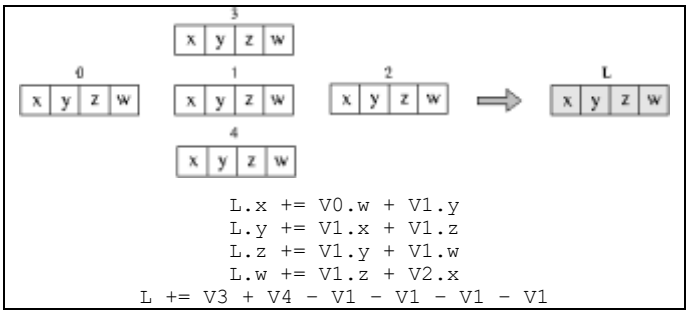

Fig. 2. Laplacian computation considering 4 gray pixels at the same time.

Figure 2 shows which input image pixels (left-hand side) are used to compute the output pixels (right-hand side). Boxes in the image represent different texture items. Each item is composed by 4 elements (*x, y, z ,w*) representing 4 gray scale pixels. The formulas at the bottom of the figure represent the computation of Laplacian for each pixel (*L.x, L.y, L.z, L.w*). Each kernel computes the Laplacian for 4 pixels at the same time accessing the memory 5 times. We run (*width/4 · height*) kernels, where (*width · height*) is the size of the source image. In the CDVS Test Model implementation Laplacian operators are applied sequentially on each pixel, whereas in our solution they are run at the same time on all image pixels.

C. *Scale-space function approximation*

For each pixel in the image, a polynomial approximation of the scale-space function is searched for a local extrema over a certain interval. The coefficients of the polynomial are obtained by computing weighted sums of the Laplacian images. The scale is the parameter value where the polynomial assumes the extrema. The pixel candidates are subject to a comparison with the adjacent 8 pixels. Those having extreme polynomial values exceeding their neighbors are kept as candidates, all others are discarded.

In our OpenCL implementation we defined two different kind of kernels. The first one computes the coefficients as linear combination of Laplacian. The second one evaluates the roots of the first derivative of the polynomial, and it also stores the minimum and maximum polynomial values (used in next steps) of each pixel. For both kernel types, (*width/4 · height*) instances work in parallel. Unlike the Gaussian and Laplacian computation, where also neighboring pixels are considered, in this case each kernel performs just one memory access to process 4 gray pixels.

D. *Coordinate refinement to subpixel precision*

The previous stages compute data that are refined as follows:

- The position of candidates (see Section II.C) is refined to sub-pixel precision using a different polynomial. Variables in this polynomial are the two values that influence the displacement from the integer pixel position.
- Displacements are detected on polynomial local maximum or minimum.
- Points are discarded if the detected displacement is larger than a threshold.

```
Input  : Coefficient, Min and Max Res, Min and Max Scale Textures
Output: Keypoints
01: maxDispl = 1.0;
02: i = getXPixelPosition ()
03: j = getYPixelPosition ()
04: A = readPixelImage (CoeffcientATexture, i, j);
05: B = readPixelImage (CoeffcientBTexture, i, j);
06: C = readPixelImage (CoeffcientCTexture, i, j);
07: D = readPixelImage (CoeffcientDTexture, i, j);
08: minRes = readPixelImage (MinResponseTexture, i, j);
09: maxRes = readPixelImage (MaxResponseTexture, i, j);
10: minScale = readPixelImage (MinScaleTexture, i, j);
11: maxScale = readPixelImage (MaxScaleTexture, i, j);
12: IsExtrema = Extrema (minRes, maxRes, minScale, maxScale);
13: if (isExtrema) then
14:     p = ComputePolynomial (A, B, C, D);
15:     deltaX = ComputeDeltaX (p);
16:     deltaY = ComputeDeltaY (p);
17:     if ((|deltaX|<=maxDispl) && (|deltaY| <= maxDispl)) then
18:         isKeypoint = true;
19:     end if
20: end if
```

Algorithm 1: Keypoint detection considering 4 gray pixels.

2 Work-groups are collections of related work-items executing on a single computation unit. Work-items in the group execute the same kernel, and share local memory and work-group barriers.

We implemented a kernel able to perform all steps listed above. The kernel processes 4 gray pixels in parallel. It reads the values computed in previous steps and select the keypoints. Finally, it stores the keypoints in a texture as described in Section II.E.

Algorithm 1 shows the implementation of the kernel that detects keypoints. Functions `getPixelPosition` (lines 2÷3) return the positions of the current texture item (4 gray level pixels). Then, in lines 4÷11, we read the corresponding coefficients, responses and scale values computed in the previous steps. The `Extrema` function (line 12) is used to figure-out whether a texture item contains candidate pixels. For each candidate, the algorithm converts its position from the scale space domain to the coordinate domain. Notice that, unlike the reference Test Model implementation of the CDVS standard, we perform all tasks described above concurrently, considering 4 gray pixels at the same time.

E. *Adapting coord inates and scale to image resolution*

Candidates are detected one octave at a time, and the analyzed images in each octave have half the size of those in the previous octave. The coordinates and scales are referred to the coordinate system of the octave in which they are detected. A further step is therefore necessary to map coordinates and scale to the resolution of the converted image [2].

Once coordinate are mapped, we need to store the keypoints. We store the keypoints in OpenCL texture, instead of saving it on temporary CPU data structures. This drastically reduces the memory access latency of the other kernels for the next computation.

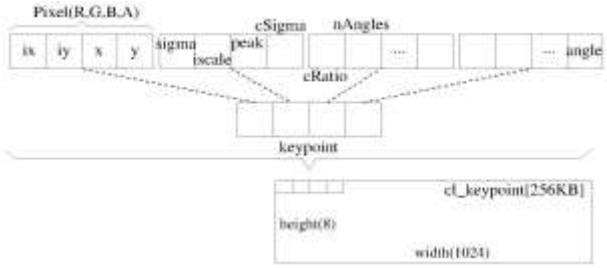

Fig. 3. OpenCL implementation: A Keypoint Structure.

Figure 3 shows the OpenCL texture used to store the computed keypoints. The maximum size of this texture is 1024 · 8. Each keypoint is stored in 4 elements, and each element is composed by 4 float numbers. Then, the total number of float elements per keypoint is 16. The first 9 float numbers (from ix to curvRatio) represent the position, the scale and others keypoints characteristics. The last 7 elements (from nAngles to the last element) represent the keypoint orientation (which will be introduced in Section III). As the maximum number of keypoints that can be stored in a row is 256, the total number of keypoints per octave that can be stored in the texture is 2048 (256 · 8). To fill the keypoint texture described above, we use the approach described in Section II.D. Essentially, we run a kernel instance for each candidate, and this kernel checks whether the candidate is actually a keypoint. In the affirmative case, the kernel has to access the keypoint structure and increment the total number of keypoints. As the keypoint structure is shared by several kernels, proper synchronization has to be adopted. We used the `OpenCL atomic_inc` function to perform atomic increments on global keypoint counters. The keypoint texture will be also used in the orientation stage (see Section III).

F. *Elimination of Duplicates*

When all octaves are completed, the next step has to eliminate interest points duplicates derived from independent processing of each octave. Each candidate detect in the previous octave q is compared with each candidate detected in octave $q - 1$. If the coordinates and the scale of the two candidates are below two given thresholds, they are subject to the following elimination process:

- If the polynomial values associated with the two candidates have opposite signs, both of them are kept.
- If the polynomial values have the same sign, then the candidate with the smallest absolute value is eliminated.

The OpenCL kernel implemented to accomplish this task works with the keypoint textures described in Section II.E. We run ($nKeypoints_q \cdot nKeypoints_{q-1}$) kernel instances where $nKeypoints_q$ and $nKeypoints_{q-1}$ are the numbers of computed keypoints of the octave q and $q-1$, respectively. To check whether a keypoint k, of the octave q, is a duplicate a kernel instance compares k with all keypoints of the $q-1$ octave.

III. Orientation Assignment

To allow rotation invariance for the subsequent feature description, a dominant orientation is assigned to each keypoint. Given an interest point detected in a Gaussian-filtered image, within an octave, with certain location and detection scale, the gradient magnitude and direction is computed for every pixel in the interest point's neighboring region. In order to accomplish this task, we implemented a kernel that computes, for each pixel, the gradient magnitude and direction. Another kernel computes an orientation histogram from the computed gradient orientations. We run ($N \cdot N$) instances of the last kernel, where N is number of neighboring pixels of the gradient texture involved in the computation. Each pixel within the circular patch is added to its nearest two histogram bins based on its orientation. In this way, histogram bin values are accumulated by the increment values of specific functions multiplied by a Gaussian weighted window with a radius of 1.32 times the detection scale (histogram smoothing). We use OpenCL kernel work-groups in order to smooth the orientation histogram. A work-group must consist of at least one work-item (kernel). The maximum number of work-items is platform dependent. The work-items within a work-group must be synchronized to share local memory with each other.

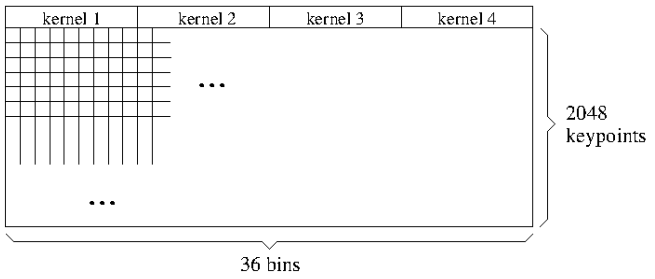

Fig. 4. Orientation histogram for our OpenCL implementation: Work-group shared memory.

Figure 4 shows the work-group shared memory; i.e., a buffer composed by (2048 · 36) integer elements, where 2048 is the maximum number of keypoints per octave, and 36 is the length of the histogram of each keypoint. Dominant orientations are determined by locating the peaks in the orientation histogram. The bin corresponding to the highest peak (as well as the bins with a value greater than 80% of the highest value) is selected as the dominant orientations of the interest point. To find the peak, 4 kernels belonging to our work-group are run together. Each kernel computes the peak on 9 histogram elements, then the first kernel of the group computes the final peak value. Once the peak is selected, a quadratic interpolation between the peak and its two neighboring bins is performed to obtain more accurate orientation.

IV. Experimental Results

The target of this section is to evaluate results accuracy (precision), and computation efficiency (speed), of our OpenCL implementations of the CDVS Detector (ALP) on embedded GPU platforms. We selected two widely used devices: The Samsung Galaxy Note 3, and the Arndale Octa board. Samsung Galaxy Note 3 embeds a CPU quad-core ARM Cortex A15, and an Adreno 330 GPU. The Arndale octa board adopts a Cortex A15 CPU, and a Mali T628 GPU. We proceed as follows. Section IV.A compares our ALP OpenCL detector with the CDVS Test Model implementation, in terms of time efficiency, on a selected number of images. Section IV.B compares the same detectors in terms of accuracy, presenting an intensive set of experiments on pairwise-matching object recognition.

A. *Efficiency*

In this section we compare detectors in terms of efficiency by selecting 12 representative images from the CTurin180 [10] test set. Table 1 compares run times of CDVS detector (ALP) Test Model (TM) implementation (CPU column) with our OpenCL implementation (GPU column) for Galaxy Note 3 and Arndale Octa board devices respectively. We measured the wall-clock time[3] between the start and the end of the kernels using OpenCL events. We wait for the end of the kernel using OpenCL function `clWaitForEvents`. Results clearly shows that our solution is up to 7 times faster than the TM one for the selected embedded devices.

In order to perform an in-depth analysis of our detector, we present a breakdown of the total time into three main stages. The first step, i.e., the Gaussian Scale Space computation stage (see Section II.A), is strongly influenced by the image size. It approximately needs the 28% of the total time, on average. The following stage, including the phases analysis in Sections II.B, II.C, II.D, and II.E, needs about 50% of the total time. The last stage, see Section III, requires the 22% of the total time on average, and strongly depends on the number of detected keypoints. Overall, detection is the most time consuming step, due to the huge number of branches (i.e., "if" statement) present in this stage.

B. *Accuracy*

In this section, we compare detectors in term of accuracy. Our benchmark set is taken from [10] and [11]. Following the CDVS standard computation scheme [2], we present pairwise matching test results. Pairwise image matching establishes correspondence between two images and it assesses whether they depict the same objects or scenes. The matching test is performed by computing the keypoints and the descriptors of two images (Query and Reference). We then compare the descriptors, i.e., computing the distance between them, in order to compute the matching score.

We work on about 2500 images, generating about 10000 matching-pairs and about 5000 nonmatching pairs.

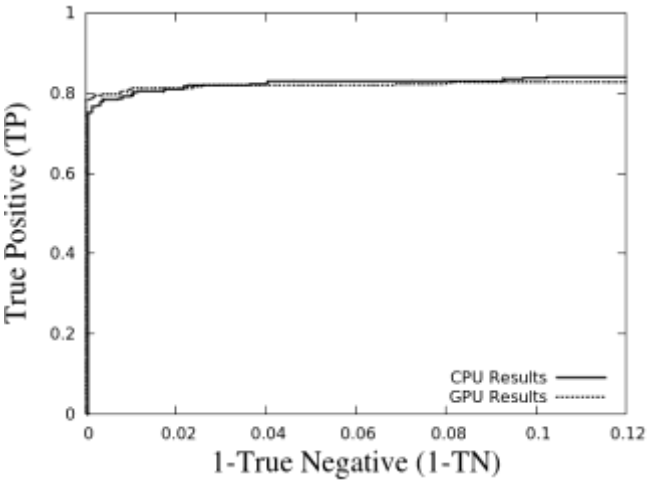

Fig. 5. Comparing GPU and CPU results using a Receiver Operating Characteristics (ROC).

Figure 5 shows the Receiver Operating Characteristic (ROC)[4] curve [12] for the CPU and GPU implementations on the same benchmark set. Even if we performed the test on both GPUs mentioned above, we report results for just one of them, because the data are identical in both cases. Essentially, the x axis reports the complemented value of the true negative matching rate (1 − *True Negative*, i.e., 1 − *TN*), while the y axis reports the true positive matching rate (*TP*).

[3] The wall clock time is the time necessary to the (mono-thread or multi-thread) process to complete the tack, i.e., the difference between the time at which the task finished and the time at which the task started. For this reason the wall clock time is also known as *elapsed time*.

[4] A ROC curve is a graphical plot which illustrates the performance of a binary classifier system while its discrimination threshold is varied. Please, see the reference for further details.

Benchmark	Galaxy Note 3			Arndale		
	CPU [ms]	GPU [ms]	Speed Up	CPU [ms]	GPU [ms]	Speed Up
Object001	460.35	80.42	5.72	461.12	112.77	4.08
Object002	471.62	73.19	6.44	477.14	100.82	4.73
Object003	469.56	71.49	6.56	467.87	102.24	4.57
Object004	530.64	94.80	5.59	532.80	132.08	4.03
Object005	472.33	92.22	5.12	470.33	118.19	3.97
Object006	467.90	94.62	4.95	468.16	129.69	3.60
Object007	441.68	101.56	4.35	440.10	133.51	3.29
Object008	433.66	96.18	4.51	433.91	115.27	3.76
Object009	430.68	101.23	4.25	433.98	129.02	3.36
Object010	439.54	100.04	4.39	440.44	136.40	3.22
Object011	432.88	96.46	4.49	432.45	124.09	3.48
Object012	451.66	92.28	4.89	448.38	118.10	3.79

Table 1: Analysis of the ALP CDVS detector. GPU times versus standard CPU times (and relative speed-ups) on the Samsung Galaxy Note 3 and the Arndale octa board embedded platforms.

Those values are plotted for several matching and non-matching thresholds varying along the curves. The graph shows a sharp knee after which it remains stable around a y-value of about 0.8. This means that, for a wide range of threshold values, the number of true positives (i.e., correct matching results) remains around 80% (y-value), while the number of wrong negatives (i.e., wrong non-matching results) stays below 12% (x-value). In other words, the graph proves that the two visual matching systems under test have similar behaviors in terms of accuracy, and the exact response (i.e., the pair is matching or non-matching) is given with a very high probability

v. Conclusions

In this paper, we present an efficient OpenCL GPU-based implementation of the CDVS ALP detector. We discuss strategies and recommendations to implement parallel CDVS algorithms on GPUs. Experimental results on standard images show that our implementations have a speed-up up to 7x over the CDVS Test Model CPU implementation. Moreover, pairwise-matching experiments clearly show that our implementation is very close to the Test Model one in terms of accuracy.

References

[1] Ling-Yu Duan and Feng Gao and Jie Chen and Jie Lin and Tiejun Huang, "Compact descriptors for mobile visual search and MPEG CDVS standardization," in ISCAS'13, 2013, pp. 885–888.

[2] MPEG, "Text of ISO/IEC CD 15938-13 Compact Descriptors for Visual Search," 2013.

[3] M. S. Mohammadi and M. Rezaeian, "Towards Affordable Computing: SiftCU a Simple but Elegant GPU-based Implementation of SIFT," International Journal of Computer Applications, vol. 90, no. 7, pp. 30–37, March 2014.

[4] Guohui Wang, Blaine Rister, and Joseph R. Cavallaro, "Workload analysis and efficient OpenCL-based implementation of SIFT algorithm on a smartphone," in Proc. IEEE Global Conference on Signal and Information Processing (GlobalSIP). IEEE, Dec. 2013, pp. 759 – 762.

[5] Seung Heon Kang, Seung-Jae Lee, and In Kyu Park, "Parallelization and optimization of feature detection algorithms on embedded GPU," 2014.

[6] Anton I. Vasilyev, Andrey A. Boguslavskiy, and Sergey M. Sokolov, "Parallel sift-detector implementation for images matching," 2011.

[7] G. Francini M. Balestri and S. Lepsoy., "CDVS: Telecom Italia's Response to CE1 - Interest point detection," Tech. Rep. ISO/IEC JTC1/SC29/WG11 Doc. M31369, MPEG-7 Video Subgroup: Compact Descriptors for Visual Search, http://wg11.sc29.org/, Geneva, Switzerland, 2013.

[8] Tony Lindeberg, "Discrete derivative approximations with scale-space properties: A basis for low-level feature extraction," Journal of Mathematical Imaging and Vision, vol. 3, no. 4, pp. 349–376, 1993.

[9] A. P. Witkin, "Scale-space Filtering," in Proceedings of the Eighth International Joint Conference on Artificial Intelligence - Volume 2, San Francisco, CA, USA, 1983, IJCAI'83, pp. 1019–1022, Morgan Kaufmann Publishers Inc.

[10] Telecom Italia, "CTurin180, http://pacific.tilab.com/www/datasets/" 2012.

[11] Computer Vision Laboratory, "Zurich Building Image Database, http://www.vision.ee.ethz.ch/showroom/zubud/index.en.html" 2003.

[12] F. Oberti, A. Teschioni, and C. S. Regazzoni, "ROC curves for performance evaluation of video sequences processing systems for surveillance applications," in IEEE International Conference on Image Processing (ICIP), Oct. 1999, pp. 949–953.