A Fast MPEG's CDVS Implementation for GPU Featured in Mobile Devices

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received July 12, 2018, accepted August 16, 2018, date of publication September 17, 2018, date of current version October 12, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2870283

A Fast MPEG's CDVS Implementation for GPU Featured in Mobile Devices

ALESSANDRO GARBO AND STEFANO QUER
DAUIN, Politecnico di Torino, 10129 Turin, Italy
Corresponding author: Stefano Quer (stefano.quer@polito.it)

ABSTRACT The Moving Picture Experts Group's Compact Descriptors for Visual Search (MPEG's CDVS) intends to standardize technologies in order to enable an interoperable, efficient, and cross-platform solution for internet-scale visual search applications and services. Among the key technologies within CDVS, we recall the format of visual descriptors, the descriptor extraction process, and the algorithms for indexing and matching. Unfortunately, these steps require precision and computation accuracy. Moreover, they are very time-consuming, as they need running times in the order of seconds when implemented on the central processing unit (CPU) of modern mobile devices. In this paper, to reduce computation times and maintain precision and accuracy, we re-design, for many-cores embedded graphical processor units (GPUs), all main local descriptor extraction pipeline phases of the MPEG's CDVS standard. To reach this goal, we introduce new techniques to adapt the standard algorithm to parallel processing. Furthermore, to reduce memory accesses and efficiently distribute the kernel workload, we use new approaches to store and retrieve CDVS information on proper GPU data structures. We present a complete experimental analysis on a large and standard test set. Our experiments show that our GPU-based approach is remarkably faster than the CPU-based reference implementation of the standard, and it maintains a comparable precision in terms of true and false positive rates.

INDEX TERMS Computer applications, concurrent computing, embedded software, image analysis, object detection.

I. INTRODUCTION

In modern scenarios (i.e., museums, exhibitions, etc.) a user, equipped with a mobile terminal, takes pictures and receives information on them in real time. The mobile terminal automatically verifies whether two images depict the same objects or the same scene. This task is usually referred as ''pairwise matching''. Matching can be performed by on-line or off-line analysis. In on-line applications, to minimize the amount of data transferred over the network and to reduce the latency time, the terminal should extract (from the picture) and deliver (to a workstation) only those data that are essential to the matching. Alternatively, the overall process, i.e., extraction and image matching, may be performed off-line directly on the device [1]. The extraction of the data essential to the matching has received specific attention by the Moving Picture Experts Group (MPEG), producing the so-called Compact Descriptors for Visual Search (CDVS) [2]. Following this standard, extracting local features from an image requires six phases, usually referred to as keypoint detection, orientation assignment, feature selection, local descriptor computation, local descriptor compression, and coordinate coding. One more step, the aggregation of local descriptors, is necessary to generate a single global descriptor. In CDVS, a descriptor is a sequence of bits which represent information about an image. A descriptor may contain information about specific areas of the image (provided by local features), and information about the image as a whole (provided by global descriptors). A local feature is a vector of values whose elements characterize a point neighborhood. Such a point corresponds to an image detail, and it is usually named ''keypoint'' or ''interest point''. Global descriptors enable the search of similar images (i.e., image retrieval), whereas local features empower pairwise matching (i.e., image matching).

The CDVS detector [3], named ALP (A Low-degree Polynomial), identifies interest points finding local extrema in the scale-space by approximating it using polynomials [4]. 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 low complexity. This process is often referred to as keypoint detection (KD). To achieve invariance of the final matching to image rotations, a canonical orientation is associated to each keypoint [5]. This stage is referred to as orientation assignment (OA). By estimating how likely any feature will be correctly matched, we may eliminate the least likely ones and pack only the most promising features into the compact descriptor. This process is often referred to as feature selection (FS). Selected keypoints are used for the local descriptors computation (LDC). Local descriptors are then compressed, subsequently encoded, and finally used in the pairwise matching phase to evaluate the matching probability. When a keypoint in one image matches a keypoint in another image, there is a high probability that the two keypoints actually correspond to the same point within the same depicted scene.

Although the above phases are extremely complex and they entail several choices and branches, they mainly manipulate images, and, for that reason, they are highly parallelizable. For this reason, the main target of this paper is to re-implement the entire process of descriptors extraction, that is, re-engineering all main CDVS stages, to efficiently run on many-cores General Purpose Graphical Processor Units (GPGPUs) supporting the OpenCL (or CUDA) language. As the work has been developed under an industrial non-disclosure agreement between Politecnico di Torino and Telecom Italia Joint Open Lab (which contributed, within the MPEG group, to define the standard itself) our main target was to obtain the best-possible behavior with a fully compliant implementation, not to modify or improved the standard itself. To this extend, one of the driving idea was to reduce CPU-to-GPU communication and interference, thus letting the CPU free to work on alternative tasks. To reach this goal, we adapted the standard to parallel processing by using the right kernel structure to implement the overall process with the maximum possible parallelism. To reduce memory accesses, optimize transfer times, decrease latencies, and efficiently distribute the workload of all OpenCL kernels, we used new approaches to store and retrieve KD, OA, FS and LDC information on proper GPU data structures. To reduce overheads, we introduced some approximation techniques to implement expensive sequential steps in the parallel environment.

More in details, our work entails the following contributions:

- We describe the standard CDVS ALP detector and we redesign it to efficiently manipulate images on many-cores general purpose graphical processor units supporting the OpenCL (or CUDA) language. This step implies a detailed analysis of the original algorithm to adapt it to parallel processing.
- We use new approaches to store FS and OA information on proper GPU data structures in order to reduce memory accesses and to efficiently distribute the workload of OpenCL (or CUDA) kernels.
- We introduce a new technique, a sort of Open-CL ''texture pagination'', to perform FS, OA and LDC. In our application, while several working kernels are dedicated to manipulate data following the standard data-flow, others are adopted to re-organize these data to make the former kernels more efficient.
- As following Amdahl's law the speedup of a concurrent program is limited by the serial part of the program itself, we introduce approximation techniques to implement expensive sequential steps in a parallel environment. For example, the standard implementation of the FS phase requires a sorting step to select the more promising keypoints. As this step cannon be efficiently parallelized in GPU, we present an alternative implementation in which we do not order keypoints and we estimate the final result. We essentially apply a bucket-sort-inspired pseudo-sorting algorithm, and we experimentally prove that our estimates reach the same final pair-wise matching accuracy, and they are more time efficient, than the original sorting strategy.
- We analyze a different CDVS flow where the FS step is performed before the OA phase. We call this strategy ''hastened feature selection'', and we prove that it is slightly more efficient than the standard approach (the ''deferred feature selection'' scheme).
- We present a complete suite of experimental results on standard benchmarks usually adopted by MPEG. On this suite we prove the consistency of our implementation on different hardware platforms. Moreover, we compare our embedded-GPU based implementation with the original one. We demonstrate that our final application is more efficient than, and at least as accurate as, the standard CPU-based implementation.

It has to be noticed that this paper is an extended version of the conference papers [6], [7], each one presenting only a few steps of the CDVS chain. The current paper, on the contrary, describes the entire CDVS implementation flow from the source images to the extraction of compact descriptors. We also extend our conference works by introducing a new methodology to perform OA using Open-CL textures and presenting the implementation of the LDC stage. We keep the entire workload and data flow within the GPU even if this may be prone to some level of inefficiency. In this way we minimize data transfers which are by nature expensive in terms of elapsed times and used memory. Moreover, we keep the CPU idle for longer periods of time, enabling the CPU to work on other tasks that may be deemed necessary on embedded and power-limited systems. All main steps are fully described from main ideas to algorithmic details. Results have been revised and reformatted, and we accurately validate our approaches reporting new data.

As far as we know, some of the previous contributions are presented for the first time, and this is the first work presenting a complete Open-CL (or CUDA) GPU-based implementation of the standard. This research has been partially sponsored by Telecom Italia S.p.A. with industrial project

"Algorithms' Optimization for Visual Analysis." It is also worthwhile recalling that the work was supported by an industrial contract with Telecom Italia Joint Open Lab. For that reason, the software and the experiments cannot be made publicly available.

A. ROADMAP

In the following description, we mainly focus on the more time consuming and complex key-operations. All others phases will be described more superficially for the sake of space. The rest of the paper is thus organized as follows.

Section II reports some comments on (and comparisons to) previous works. Section III presents background notions on the CDVS standard and GPU. Section IV, V, VI, and VII describe our implementation of the main phases. Section VIII illustrates experimental results on standard benchmarks, highlighting performance and precision accuracy. Finally, Section IX concludes the paper with some summarizing remarks.

II. RELATED WORKS AND COMPARISONS

Due to the integration of multiple heterogeneous processing units, programmers can make use of processors with various features. However, data transfer and partitioning schemes appear as challenging tasks. The followings are several attempts to adopt heterogeneous computing for several applications close to the one we are dealing with in the paper.

Wang *et al.* [8] implement some major steps of the Scale-Invariant Feature Transform (SIFT [5]) using both serial C++ code and OpenCL kernels targeting mobile processors. Based on profiled results of different work-flows, they partition SIFT between the CPU and the GPU to better exploit the parallelism, and to minimize buffer transfer times. Suárez *et al.* [9] introduce a CMOS vision sensor to extract the Gaussian pyramid with an energy cost lower than the one of conventional solutions. The chip, manufactured in a 0.18 μm CMOS technology, consists of an arrangement of $[88 \times 60]$ processing elements. These units capture images, and they perform concurrent parallel processing right at pixel level. Leyva *et al.* [10] concentrate on hardware architectures to speed-up the computation of the feature descriptor vector in SIFT. The architectures could be time-optimized or memory-optimized, and they computed a feature descriptor vector of 27 elements, starting from a keypoint neighborhood of $[15 \times 15]$ pixels, in 649 or 874 clock cycles, respectively. The process involves several steps, including complex ones such as vector normalization. Lee *et al.* [11] present a sequential implementation of a two-stage FS based on the CDVS Test Model (TM). They significantly reduce run times while maintaining the original matching and retrieval accuracy. This implementation is the closest one to ours in the literature, and we use some of their ideas to implement our version of the FS module. Nevertheless, our implementation is faster and several steps have been further optimized. Zhang *et al.* [12] accelerate the CDVS extracting process on a multi-core ARM processor. They implement a NEON SIMD-based data level parallelism and a Pthread-based multi-thread parallelism scheme for mobile devices. They achieve significant speed-up in the keypoint detection and in the local descriptor computation stages. Arndt *et al.* [13] present a heterogeneous implementation of the Histograms of Oriented Gradients algorithm targeting the CPU-clusters and the GPU of the Samsung Exynos 5 Octa 5422. The authors present different strategies to generate the best partitioning scheme, and they analyze the computational capabilities as well as the power consumption of the individual processing units. Doush and AL-Btoush [14] develop an automatic banknote recognition, to classify Jordanian currency to the correct class. The application uses SIFT, and it runs SIFT on smartphone devices. The authors also compare a SIFT approach based on colored images with the one based on gray tones. Lee *et al.* [15] propose an efficient scheme to optimize SIFT for a mobile GPU. They analyze the conventional scale-space construction step in the SIFT generation, finding that reducing the size of the Gaussian filter and the scale-space image leads to a significant speed-up with only a slight degradation of the quality of the features. Based on this observation, they modify SIFT for real-time execution. They also obtain additional speed-up by efficiently using both the CPU and the GPU available on the mobile device. Duan *et al.* [16] presents a fast CDVS encoder implemented using hybrid GPU-CPU computing. The authors shift all computation-intensive and parallel-friendly modules to the GPU platform. They also incorporate the CDVS encoder with deep learning based approaches. A comparison with other state-of-the-art visual descriptors shows that they achieve significant speed-up compared with algorithms running on pure CPU platforms while obtaining similar results for image retrieval and matching accuracy.

To sum up, many works propose strategies to shift on a GPU architecture only the most expensive and highly parallelizable algorithmic steps. In those schemes data are usually transferred several times forward-and-backward between the CPU and the GPU. This sort of overhead may become a bottleneck in several applications and it makes the overall algorithmic flow intrinsically sequential. On the contrary we strive to keep the entire workload and data flow within the GPU even if this may be prone to some level of inefficiency. In this way, we minimize data transfers which are by nature expensive in terms of elapsed times and used memory. Moreover, we keep the CPU cores idle for longer periods of time enabling them to work on other tasks that may be deemed as necessary on embedded and power-limited systems.

As a final remark, notice that Francini *et al.* [17] present a method to select features based on characteristics computed by the keypoint detection process. They claim that in the case of detectors based on Gaussian Scale Space theory [18], the most important characteristics are the keypoint location, scale, absolute value of the detected extreme, and the orientation. They prove that each of these characteristics has a strong impact on the probability of having correct feature match. This methodology has been inserted into the MPEG-CDVS standard [19], and we base our KD, OA, and LDC phases exactly on this algorithm.

III. BACKGROUND

A. TERMINOLOGY AND NOTATION

In the rest of the paper, we will use the following notation and terminology:

- KD, OA, FS, LDC, MP indicate keypoint detection, orientation assignment, feature selection, local descriptors computation, and matching probability, respectively.
- As kernels are organized as 2D-matrices of threads, we use square brackets to indicate the number of threads run by a kernel, e.g., $[width \times height]$.
- k indicates a keypoint, and K a keypoint set (with $k \in K$). f indicates a feature, and F a feature set (with $f \in F$). Subscripts indicate keypoint (feature) attributes, i.e., matching probability (k_{mp}), orientation (k_α), etc. In the CDVS terminology (see Section III-B) a keypoint k becomes a feature f when the former is enriched with the orientation attribute k_α.
- O (S) indicates the octave set (filter set), $|O|$ ($|S|$) its cardinality, and o (s) a specific octave (filter).
- W and H indicate the image width and height, thus W_o and H_o the image width and height at octave $o \in O$.
- G (L) indicates Gaussian images (Laplacian) within the Gaussian Scale Space (GSS). $G_{o,s}$ ($L_{o,s}$) indicates a Gaussian image (Laplacian) at octave $o \in O$ and with filter $s \in S$.
- p indicates a pixel and t a texel. This terminology follows the one introduced by Doggett [20], where individual elements in the texture map are called texels (from "TEXture ELements") to differentiate them from the screen pixels. T indicates an OpenCL texture.

B. THE CDVS STANDARD

The compact descriptor of an image is composed of two main elements, that is, a selected number of compressed local descriptors (representing confined picture areas) and a single global descriptor (representing the whole image). Fig. 1 illustrates how these two elements are produced by a series of processing steps (the ones already listed right in the introduction) starting from an input image.

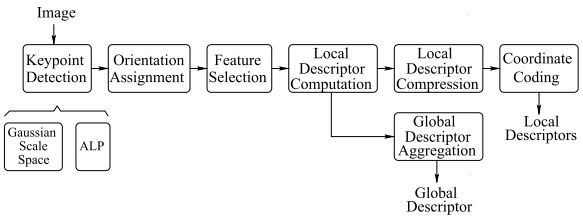


FIGURE 1. Descriptor extraction pipeline: Computing the global and the local descriptors from the source image.

The global descriptor is finally used for image retrieval, while local descriptors are used to check image correspondence [21]. Notice that keypoint detection, orientation assignment, feature selection, and local descriptor computation are the most CPU time and memory expensive steps. At the same time, they consider information related to neighboring pixels, thus they are highly parallelizable. The objectives of these steps are the following:

- The keypoint detection phase (KD) can be further divided into the Gaussian Scale Space computation and the identification of Low Polynomial degree (ALP) [3] keypoints. Following Witkin [22], the Gaussian Scale Space (GSS) is generated by a sequence of Gaussian filters of increasing size and different Gaussian's standard deviation. For each pixel in the image, ALP generates a polynomial approximation of the scale-space function and it searches local extrema over a certain interval. The coefficients of the polynomial are obtained by computing weighted sums of the Laplacian of Gaussian 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 the others are discarded.
- During orientation assignment (OA), 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 with respect to image rotation. Notice that the standard describes this step within the keypoint detection phase. We detail it separately only because its efficient implementation is part of our contribution.
- Feature selection (FS) chooses a limited number of keypoints to improve the quality of the final matching. The selection is based on the matching probability (MP) associated to each keypoint. In turn, the MP of a keypoint is computed by the standard by applying several quantification steps to keypoint information evaluated during the KD phase.
- Local descriptor computation (LDC) calculates descriptors for all keypoints selected during the previous phase. Local descriptors are the used in the pairwise matching phase.
- Local descriptor compression is a 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 is the compression of the coordinates of the selected keypoints to improve storage efficiency.
- Global descriptor compression consists in the aggregation of local descriptors, to form a single global descriptor. Global descriptors enable image retrieval, that is, the search for similar images. We will not concentrate on this step as we mainly focus on image matching.

C. GPU ARCHITECTURES

Initially developed for real time and high-definition 3D graphic applications, graphics processing units (GPUs) have

recently gained attention for high performance computing applications (general purpose GPU or GPGPU). Indeed, the peak computational capabilities of modern GPUs exceeds the one of top-of-the-line central processing units (CPUs). GPUs are highly parallel, multi-threaded, many-core units and have been recently used in a plethora of different applications and they are becoming more and more invasive in every-day life [23]–[28]. GPUs are SIMT (single-instruction, multiple-threads) architectures, i.e., the same instruction is executed simultaneously on many data elements by different threads. They are especially well-suited to address problems that can be expressed as data-parallel computations.

The execution starts with a host (CPU) execution. When an OpenCL or a CUDA kernel function is invoked, the execution is moved to a device (GPU) and a grid is launched on the GPU. A grid represents a set of blocks, and each block contains up to N threads, where N coincides with the number of cores of the architecture. When the grid of threads of a kernel complete its execution, the corresponding kernel terminates, the execution continues on the host until another kernel is invoked. When more kernels are run in parallel they are pipelined by the GPU. When a kernel is launched, each processor runs one block by executing groups of parallel threads (named ''warps'' in the CUDA terminology). Threads composing a warp start together at the same program address. Nevertheless, they are free to branch and execute independently. As thread blocks terminate, new blocks are launched on the idle multiprocessors.

Threads have access to data stored on multiple memory spaces. We can distinguish several types of memory spaces (from the one with the smallest latency time on):

- Constant memory (for read-only constant data), registers, and private local memory space. This is local to each thread.
- Shared memory spaces. This is accessible only by threads in the same block.
- Global memory. This is visible by all grid threads and it is accessible in read-and-write mode by all threads but with larger latency times.

With CUDA 3.0, threads of different blocks cannot communicate with each other explicitly but they can share their results through a global memory. If threads of a warp diverge when executing a data-dependent conditional branch, then the warp serially executes each branch path. This leads to poor efficiency.

Another important aspect within GPU architectures is the copy strategy, i.e., the method used to bring data from CPU to GPU memory space and vice-versa. In discrete devices, such as desktops and servers, this basically translates into copying memory from the system DRAM, through PCI, towards the on-board low-latency memory space of the graphics adapter. In case of embedded platforms, most GPUs implement local memory through global memory. In this case, local memory should not be used as a software-managed cache for performance. As a final remark, notice that both CUDA and OpenCL programming models specify alternatives to avoid explicit memory transfers and unnecessary buffer replications, such as CUDA UVM (Unified Virtual Memory) and OpenCL 2.0 SVM (Shared Virtual Memory). However, these approaches introduce CPU-GPU memory coherency problems when accessing the same shared memory buffer, so that avoiding copy engines does not necessarily lead to performance improvements.

IV. KEYPOINT DETECTION (KD)

As described in Section III-B, the keypoint detection phase can be further divided into the Gaussian Scale Space computation and the identification of keypoints using ALP. We will analyze both phases into details in the following paragraphs.

A. GAUSSIAN SCALE SPACE

The Gaussian Scale Space computation is the first CDVS operation working on the input image. Algorithm 1 illustrates this phase as implemented by the standard.

Algorithm 1 GSS Computation

Require: G, S
Ensure: $G_{O,S}, L_{O,S}$

1: **GSS**
2: $G_{1,0} = G$
3: **for** $o = 1$ to $|O|$ **do**
4: $(W_o, H_o) =$ **ImageRange** $(G_{o,0})$
5: **for** $s = 1$ to $|S|$ **do**
6: $G_{o,s} =$ **GaussianFilter** $(G_{o,s-1}, W_o, H_o, S_s)$
7: **end for**
8: **for** $s = 1$ to $|S|$ **do**
9: $L_{o,s} =$ **Laplacian** $(G_{o,s})$
10: **end for**
11: **if** $(o < |O|)$ **then**
12: $(G_{o+1,0}) =$ **Reduction** $(G_{o,|S|-1})$
13: **end if**
14: **end for**

Function GSS receives as input parameters the gray tone image G and the set of Gaussian filters S of cardinality $|S|$. Its target is to compute and to return the Gaussian image $G_{O,S}$ and the Laplacian $L_{O,S}$ evaluated during all octaves O and all filters S. $G_{o,s}$ ($L_{o,s}$) indicates the Gaussian image (Laplacian) computed at octave o and filter s. Function GSS iterates $|O|$ times with images of decreasing size. For each octave $o \in O$ (line 3), function GSS first computes a new size image (W_o, H_o). Then, it filters the current image through $|S|$ Gaussian filters and $|S|$ Laplacian operators. The first sequence of Gaussian filters (lines 5–7) starts from the gray scale input image G assigned to $G_{1,0}$ in line 2. All following ones start from a rescaled image coming from the previous octave and generated by function Reduction (lines 11–13). The Laplacian operator Laplacian (line 9) is applied to each Gaussian filtered image previously generated.

To implement Algorithm 1 on a GPU, and more specifically functions GaussianFilter, Laplacian, and Reduction, we proceed as follows.

1) GAUSSIAN FILTERING

In our scenario, we work with gray scale images. An image, on the other hand, can be manipulated at its best using GPU textures. We use RGBA OpenCL cached textures T where every texel t includes 4 values (x, y, z, w). As a consequence, we encode 4 gray level pixels p in one texel t (see Doggett [20]) of the texture T, reducing the number of memory accesses.

This choice is motivated by the following considerations. OpenCL *cl_image* data structure, as well as CUDA textures, have a caching support, whereas global memory does not have it (at least traditionally[1]). Several papers analyzed the performance difference between textures and global memories (see for example Hakura and Gupta [29] and Wong *et al.* [30]). Level 1 and Level 2 caching has also been proved to be quite beneficial (especially during reading operations) by several authors (see for example Doggett [20]). In practice, we use cached textures in several parts of our implementation (but in those cases in which we need a proper synchronization among threads, see for example Section V-A and Section VI-B). To be concrete, we run some experiments on the keypoint extraction phase. We verified that global memory is from 50% to 70% slower than cached textures in our approach.

The Test Model (TM) implementation of CDVS [16] performs the Gaussian Filtering of Algorithm 1 using a sequence of one 1D horizontal filter and one 1D vertical filter. On the contrary, we implement this process as a sequence of two horizontal filters using 4 kernels which implement the following operations:

1) Filter the image T with a 1D horizontal Gaussian.
2) Take the transpose T^T of T.
3) Convolve with the same 1D horizontal Gaussian filter the image T^T.
4) Transpose one more time the result from T^T to T.

With a source image of size $[width \times height]$ pixels, we run $[\frac{width}{4} \times height]$ threads for the horizontal filtering operations (steps 1 and 3) and $[\frac{width}{4} \times \frac{height}{4}]$ threads for the transposition (steps 2 and 4). Every thread of the 4 kernels sequentially manipulates 4 consecutive horizontal gray tone pixels, i.e., (x, y, z, w), within the RGBA texel t. Those pixels represent the atomic pieces of information loaded within the thread. We decided to use just horizontal filters due to the gray pixel coding policy adopted inside the texture texels t. Each horizontal filter involves a number of pixels which depends on the filter size. CDVS uses a set of 4 Gaussian filters in sequence with size 15, 15, 21, and 27. For example, to implement a Gaussian filter of size equal to 15, the standard approach requires 15 memory accesses for each pixel. With our strategy, to upload all pixels required by the same filtering operation, we read only 5 RGBA texels within each thread. Those 5 texels store everything needed to sequentially manipulate 4 gray tone pixels in the same thread. This process is represented in Fig. 2.

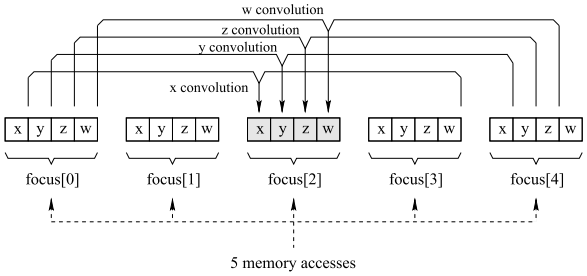


FIGURE 2. Gaussian Filtering: Performing Convolution using OpenCL kernels.

Although the distance of each keypoint to the image center is used to weight keypoints in the FS stage, and keypoints around the image boundary are rarely selected for image retrieval or image matching, the CDVS standard requires "padding" during convolution. A straightforward implementation of this process implies concurrent code including branches. Such a code, by definition, is not SIMT compliant and therefore it may be quite inefficient. To avoid such a problem, Algorithm 2 shows a solution in which branching is restricted to the first and the last two texel columns.

At the same time, all internal columns present the same branching flow. Moreover, notice that, as we use only horizontal filters, padding must be implemented only on the left and right borders. Thus, our coding policy reduces the overall working load.

Function Conv_{15} follows the process of Fig. 2 for the first and second filters of size 15. It receives the Gaussian image at octave o and filter $s-1$ (i.e., $G_{o,s-1}$), the width (W_o) and height (H_o) of the image at that octave, and the set of filters (S). It generates the Gaussian image at the octave o and filter s. In line 2, variables i and j are used to store the thread indices, i.e., the texel position within the current Gaussian image $G_{o,s-1}$. When the current pixel is not on the image border (i.e., $i > 1$ and $i < (W_o - 2)$), following Fig. 2, the four pixel data sets *focus*[0], *focus*[1], *focus*[3], and *focus*[4] are read (by function ReadItem) from the current Gaussian image $G_{o,s-1}$. On the contrary, when the current pixel is on the image border (i.e., $i = 0$ or $i = (W_o - 1)$) or just on the inner frame (i.e., $i = 1$ or $i = (W_o - 2)$), one or more pixels are obtained with padding (lines 9-10, 13, 21-22, and 25). Padding duplicates one gray tone in all texel components. Function UnrollConvolution finally computes $G_{o,s}$ by optimizing the convolution computation through loop unrolling.

To further reduce branching, it is possible to use 5 kernels to manipulate different sections of the image. In this scheme the first kernel manipulates texels within the first column, the second one pixels belonging to the second column, the third one the central image section, the fourth one pixels belonging to the last but one column, and the fifth kernel pixels within the last column. These 5 kernels would run

[1] As far as we know, caching has been introduced for global memories starting from CUDA-2.0. Anyhow, we do not take into consideration this improvement in our implementation as the hardware platforms we consider do not support it.

Algorithm 2 Edge Padding During Gaussian Filtering on a Texture With Size 15 (the First or the Second One). Notice That, With This Filter Size, the Situation Is Exactly the One Depicted in Fig. 2, Where There Are Just 2 Texels on the Right and on the Left of the Analyzed Texel

Require: $G_{o,s-1}$, W_o, H_o, S
Ensure: $G_{o,s}$

1: **Conv**$_{15}$
2: $(i, j) =$ **GetGlobalId** ()
3: $focus[2] =$ **ReadItem** $(i, j, G_{o,s-1})$
4: **if** $(i > 1)$ **then**
5: $\quad focus[0] =$ **ReadItem** $(i-2, j, G_{o,s-1})$
6: $\quad focus[1] =$ **ReadItem** $(i-1, j, G_{o,s-1})$
7: **else**
8: $\quad$ **if** $(i == 0)$ **then**
9: $\quad\quad$ **for** $j \in \{x, y, z, w\}$ **do** $focus[0].j = focus[2].x$
10: $\quad\quad$ **for** $j \in \{x, y, z, w\}$ **do** $focus[1].j = focus[2].x$
11: $\quad$ **else**
12: $\quad\quad focus[1] =$ **ReadItem** $(i-1, j, G_{o,s-1})$
13: $\quad\quad$ **for** $j \in \{x, y, z, w\}$ **do** $focus[0].j = focus[1].x$
14: $\quad$ **end if**
15: **end if**
16: **if** $(i < (W_o - 2))$ **then**
17: $\quad focus[3] =$ **ReadItem** $(i+1, j, G_{o,s-1})$
18: $\quad focus[4] =$ **ReadItem** $(i+2, j, G_{o,s-1})$
19: **else**
20: $\quad$ **if** $(i == (W_o - 1))$ **then**
21: $\quad\quad$ **for** $j \in \{x, y, z, w\}$ **do** $focus[3].j = focus[2].w$
22: $\quad\quad$ **for** $j \in \{x, y, z, w\}$ **do** $focus[4].j = focus[2].w$
23: $\quad$ **else**
24: $\quad\quad focus[3] =$ **ReadItem** $(i+1, j, G_{o,s-1})$
25: $\quad\quad$ **for** $j \in \{x, y, z, w\}$ **do** $focus[4].j = focus[3].w$
26: $\quad$ **end if**
27: **end if**
28: $G_{o,s} =$ **UnrollConvolution** $(focus, S)$

a number of threads equal to $[1 \times height]$, $[1 \times height]$, $[(width/4) - 4 \times height]$, $[1 \times height]$, and $[1 \times height]$, respectively. Algorithm 2 can be modified following this strategy, such that all running threads would be completely SIMT complaint.

To analyze the best trade-off between the number of code branches allowed and the number of kernels that have to be run, we implemented and compared three versions of this procedure: The one with one single kernel, the one presented in Algorithm 2, and the one with 5 kernels. If we consider the intermediate solution, i.e., the one represented by Algorithm 2, about 20–25% of the time is spent in padding and about 75–80% by function **UnrollConvolution** (line 28). From the one hand, without stencil code optimization this last function would be from 4 to 5 times slower. From the other one, with one single kernel padding is about 50% slower and it is about 5–10% faster with 5 kernels. As the two versions with 2 and 5 kernels are almost equally efficient, with no clear winner (as the majority of the time is spent to unroll the convolution, anyhow), we consider Algorithm 2 as our reference implementation.

To transpose T into T^T (and vice-versa) each thread manipulate 4 RGBA pixels vertically placed, and it generates the same number of output pixels. As represented in Fig. 3, each thread identifies the area of the input texture on which it has to work, using its thread index (i, j), and it identifies the area on the output texture, using the transpose $(I = j, J = i)$.

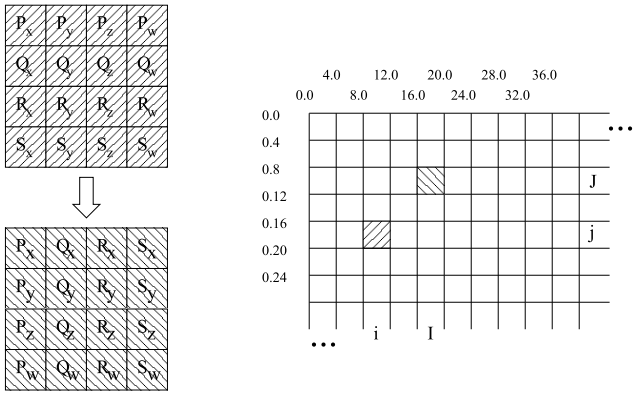


FIGURE 3. Transpose operation: From (i, j) on the input texture to $(I = j, J = i)$ on the output texture.

Using 4 sequential kernels to perform filtering reduces the number of read operations (R_t) with respect to the standard case which uses 2 kernels to perform a horizontal and a vertical filter. For example, with filters of size equal to 15, the standard procedure performs 5 read operations for the horizontal filter, and 15 for the vertical one, i.e., a total of $R_t = 5 + 15 = 20$ read operations. At the same time, our solution with 4 kernels implies 5 accesses for each convolution and 1 access for a group of 4 transpositions (that is 0.25 accesses every single texel), i.e., $R_t = 5 + 0.25 + 5 + 0.25 = 10.5$ read operations. Furthermore, it is important to remind that the GPU caching infrastructure stores a pixel t and its neighbor (after it has been read) within the hardware cache, thus reducing latency times of all subsequent accesses by other threads.

2) LAPLACIAN

The Laplacian operator (useful to highlight regions of rapid intensity change) is applied to the images that have first been smoothed by a Gaussian filter. Following Algorithm 1 the **Laplacian** function takes a single gray-level image as input and produces a Laplacian of Gaussian image as output. We implement an OpenCL kernel with $[\frac{width}{4} \times height]$ threads, which it is able to apply the Laplacian operator sequentially to the 4 gray pixels encoded in t. Fig. 4 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 texels. The standard approach does not perform padding as it rules-out all pixels of the image frame from the Laplacian computation. In our case, threads compute the Laplacian even on the

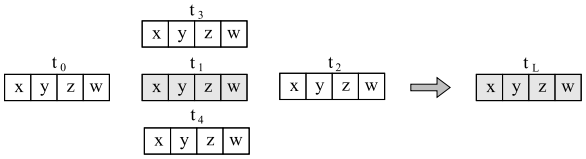


FIGURE 4. Laplacian computation considering 4 gray pixels at the same time.

frame to avoid kernel branches. Nevertheless, as our function **ReadItem** of Algorithm 2 returns (0, 0, 0, 0) on the frame, the result we obtain is exactly the one returned by the standard procedure. The formulas

$$\begin{aligned} L_{o,s}[1].x &+= G_{o,s}[0].w + G_{o,s}[1].y \\ L_{o,s}[1].y &+= G_{o,s}[1].x + G_{o,s}[1].z \\ L_{o,s}[1].z &+= G_{o,s}[1].y + G_{o,s}[1].w \\ L_{o,s}[1].w &+= G_{o,s}[1].z + G_{o,s}[1].x \\ L_{o,s}[1] &+= G_{o,s}[3] + G_{o,s}[4] - 4 \cdot G_{o,s}[1] \end{aligned} \quad (1)$$

represent the computation of Laplacian for each encoded gray pixel ($L_{o,s}[1].x$, $L_{o,s}[1].y$, $L_{o,s}[1].z$, $L_{o,s}[1].w$) given the adjacent texels ($G_{o,s}[0]$, $G_{o,s}[2]$, $G_{o,s}[3]$, and $G_{o,s}[4]$) from the correspondent Gaussian texture. Each thread computes the Laplacian for 4 pixels at the same time accessing the memory only 5 times.

3) REDUCTION

Following Algorithm 1, the $G_{o,|S|-1}$ image (i.e., the second to last filtering Gaussian from the GSS pyramid) is down-scaled by a factor of two by procedure **Reduction**. This down-scaling procedure creates the next octave, and it is performed by a dedicated kernel running a matrix of $[\frac{width}{8} \times \frac{height}{2}]$ threads. Given our texture texel t the kernel reads two consecutive texels on the same row $t'(t'.x, t'.y, t'.z, t'.w)$ and $t''(t''.x, t''.y, t''.z, t''.w)$, and it generates a single pixel $t(t'.x, t'.z, t''.xt''.z)$ skipping odd rows. The result of this reduction becomes the input $G_{o+1,0}$ for the next octave as represented by the pseudo-code of Algorithm 1.

B. THE ALP DETECTOR

The target of ALP is to find all keypoints k, and to return the entire keypoint set K_O (with $k \in K_O$), given the Laplacian L_O at the current octave. This process is described in Algorithm 3.

For each octave $o \in O$ (line 2), and for each pixel within the image available at the current octave $p \in [1, W_o \cdot H_o]$ (line 3), ALP generates a polynomial of degree 3, i.e., $\psi_o[p]$. This polynomial is generated (line 5) as a linear combination of the 4 Laplacian values $L_{o,f}$ (with $f \in [1, F]$) forming the octave. Given $\psi_o[p]$, the roots of its first derivative ($\psi'_o[p]$, line 6), are the minimum S_o^m and the maximum S_o^M values (scale). Evaluating $\psi_o[p]$ in S_o^m and S_o^M gives the minimum $R_o^m[p]$ and the maximum $R_o^M[p]$ values (response), respectively. All those data are stored in different textures in a position

Algorithm 3 Keypoint Detection The 4 Coefficients of ψ_o and All Information Within S_o^m, S_o^M, R_o^m e R_o^M Are Encoded in a RGBA Texture With 4 Components

Require: L_O
Ensure: K_O

1: **ALP**
2: **for** $o = 1$ to $|O|$ **do**
3: **for** $p = 1$ to $(W_o \cdot H_o)$ **do**
4: $(R_o^m[p], R_o^M[p]) =$ **Init** $(L_{o,1}, L_{o,F})$
5: $\psi_o[p] =$ **Coeff** (L_o)
6: $(S_o^m[p], S_o^M[p]) =$ **Root** $(\psi'_o[p])$
7: $(R_o^m[p], R_o^M[p]) = \psi_o[p]\ (S_o^m[p], S_o^M[p])$
8: $K_o =$ **Detect** $(\psi_o[p], S_o^m[p], S_o^M[p], R_o^m[p], R_o^M[p])$
9: **end for**
10: $(K_{o-1}, K_o) =$ **Duplicate** (K_{o-1}, K_o)
11: **end for**

corresponding to the one of the pixel p. This technique enables a quick retrieval of all keypoint information by all kernels that have to perform their computation. The minimum and maximum values of ψ_o become keypoint candidates if they represent minimum and maximum values around a pixel p. Function **Detect** (line 8) performs a selection process among the keypoints generated for each pixel. Once all pixels are manipulated, function **Duplicate** compares all keypoints generated during two consecutive octaves (i.e, K_{o-1} and K_o), and it retains only the strongest keypoint within each couple of duplicated keypoints.

1) SCALE SPACE APPROXIMATION AND EXTREMA DETECTION

ALP finds keypoints during each octave and it merges all results during the last iteration. To extract keypoints we use several kernels implementing the steps included in the CDVS standard. The number of kernels has been selected as the best possible trade-off between precision and efficiency. A few of them are used to run initialization operations. Others are used to find, re-arrange, and gather keypoints based on their octave.

To implement Algorithm 3 on a GPU, we use 5 different OpenCL kernels.

The first one (function **Init**, line 4) initializes response textures R_o^m and R_o^M with the lower and the upper Laplacian texture $L_{o,1}$ and $L_{o,F}$.

The second one (function **Coeff**, line 5) computes the linear combination ψ_o of 4 Laplacian coefficients for every pixel p.

The third kernel (function **Root**, line 6) evaluates the roots S_o^m, and S_o^M of the first derivative ψ'_o of the polynomial. Moreover, it finds the minimum and maximum values of those roots $R_o^m[p]$, and $R_o^M[p]$. Unlike the Gaussian and Laplacian computation, where also neighboring pixels are considered, in this case each kernel (running a matrix of $[\frac{width}{4} \times height]$ threads) performs just one memory access to process 4 gray pixels p.

All threads within kernel **Init**, **Coeff** and **Root** read all required pixel information from different textures storing intermediate data. These data are manipulated by a fourth kernel, that is **Detect**. This kernel considers only pixels with minimum or maximum local polynomial values (i.e., those pixels having values R_o^m and R_o^M exceeding the ones of their pixel neighbors), and for each of them:

- It refines the position of those candidates to sub-pixel precision using a different polynomial ω. This polynomial ω is a linear combination of the Laplacians of the area of dimension q around the examined pixel p and the root of $\psi'_o[p]$. Using the ALP standard, from ω it is possible to compute the ratio τ (the squared trace to the determinant of the Hessian), and the value of Δ_x and Δ_y. These variables are the two values that influence the displacement from the integer pixel position (I_x, I_y).
- It initializes σ as the root $S_o^m[p]$ or $S_o^M[p]$ of the polynomial $\psi'_{o,p}$ with the generation of $R_o^m[p]$ or $R_o^M[p]$ local minimum or maximum. It defines the peak ρ as the reference value $R_o^m[p]$ or $R_o^M[p]$. Moreover, it specifies the scale δ^f (with $f \in (1 \dots F)$) as the value of the Gaussian filter S_σ closer to $S_o^m[p]$ or $S_o^M[p]$.
- It computes the keypoint curvature σ_{curv} based on the second derivative of ψ_o.

Note that candidates are detected one octave at a time, and the analyzed images in each octave have a quarter of 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. Therefore a further step performed by kernel **Detect** is to map coordinates and scale to the resolution of the converted initial input image dimension [31] (coordinated domain). Once coordinates are mapped, we need to store the keypoints K_o of all octaves O into K_O.

2) ALP KEYPOINTS

Instead of storing keypoints on temporary CPU data structures, we store them in OpenCL textures. Fig. 5 shows all internal fields of this OpenCL texture.

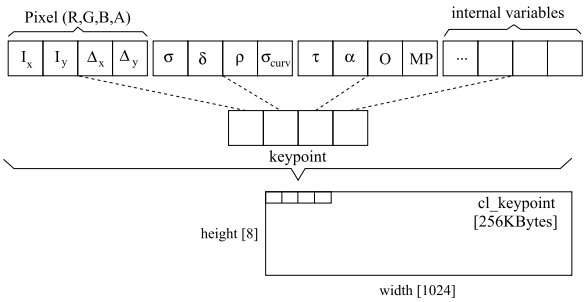


FIGURE 5. OpenCL implementation: A keypoint/feature structure.

3) DUPLICATES

In the last row of Algorithm 3 the OpenCL kernel **Duplicate** works on (K_{o-1}, K_o) represented by the keypoint textures we have just described. If keypoints are meaningful, it may happen that they are captured by different octaves $o \in O$. The procedure compares each keypoint in K_{o-1} with all keypoints in K_o. For each pair, first CDVS computes the distance between the two keypoints and the difference between their scales. Then, it compares these two values with two different thresholds. If both values (distance and scale difference) are below the threshold, the keypoint with the smallest absolute value of ρ is canceled, otherwise both keypoints are maintained.

V. FEATURE SELECTION (FS)

As represented in Fig. 1, keypoint detection (KD) is followed by orientation assignment (OA), and then by feature selection (FS). In this process, each keypoint ($k \in K$) becomes a feature ($f \in F$) when the orientation k_α is added to the original keypoint. Nevertheless, our GPU algorithm is designed in such a way that OA and FS can be swapped with some memory and time advantage. As a consequence, in the following two subsections, we present two feature selection algorithms. In the first one, which we call "deferred FS" scheme (see Section V-A), FS is performed after OA, as in the standard. In this case, our main contribution consists in adopting a GPU-oriented approximated sorting algorithm inspired by bucket-sort instead of an exact sorting algorithm. In the second one, which we call "hastened FS" scheme (see Section V-B), the FS phase is anticipated and performed before OA. The last part of this section (Section V-C) focus on one important CDVS sub-step, namely the matching probability phase.

A. DEFERRED (STANDARD) FS

In the standard scheme, FS is computed following the pseudo-code reported in Algorithm 4. The procedure receives the set of all keypoints at all octaves, K_O, and a quantification set Q. It generates the set of features at all octaves F_O, a threshold T_H, and all local descriptors at all octaves D_O.

Algorithm 4 Deferred (Standard) Approach: FS Is Performed After OA

Require: K_O, Q
Ensure: F_O, T_H, D_O
1: **DeferredFS**
2: $F_O =$ **OrientationAssignment** (K_O)
3: $(F_O, H) =$ **MP** (F_O, Q)
4: $T_H =$ **ComputeThreshold** (H)
5: $D_O =$ **LocalDescriptor** (F_O, T_H)

To do that, the CDVS standard proceeds as follows. Function **OrientationAssignment** (line 2) manipulates all keypoints (K_O) found by the KD module, and it generates a set of features (F_O). The set F_O may have a cardinality larger than the one of K_O, i.e., $|F_O| \geq |K_O|$, as the orientation assignment step duplicates all keypoints with more than one possible orientation. Then, for each feature $f \in F_O$ generated by the OA module, CDVS computes its matching probability (line 3).

Notice that in this case, function MP works on the entire set of features F_O, considering their orientations f_α directly when computing their matching probability f_{mp}. As a consequence, MP has to manipulate a quite large set of objects. Function MP will be analyzed into details in Section V-C. The list of features, enriched with the matching probability, is then sorted by function ComputeThreshold (line 4). The standard, running on CPU platforms, sorts features using a state-of-the-art sorting algorithm. Even if the number of features, for every octave $o \in O$, is upper-bounded by $(2 \cdot 10^3)$, sorting keypoints adopting a standard sorting algorithm may be quite inefficient on a GPU-based SIMT architectures [32]. Moreover, note that the CDVS standard sorts keypoints only to select the most important features, not to obtain a complete and global order among them. As a consequence, we apply an approximated sorting algorithm inspired by bucket-sort. This guarantees low time complexity with practically no penalty in terms of accuracy. It proceeds as follows.

The histogram H, returned by function MP, represents a pre-defined number of classes. Each class stands for a range interval within the minimum and maximum values of f_{mp}. This number of classes is selected as a function of the desired approximation. As the histogram's maximum number of intervals is 10^3, and $f_{mp} \in [3, 3 \cdot 10^{-3}]$, the size of each class is $[3 \cdot 10^{-6}]$. In this situation, the first class contains features having $f_{mp} \in [0, 3 \cdot 10^{-6}]$, the second class features having $f_{mp} \in]3 \cdot 10^{-6}, 6 \cdot 10^{-6}]$, etc. For each class (or bin), we define a counter representing the number of features having f_{mp} in the corresponding interval. As a consequence, once all f_{mp} values have been computed, each feature is assigned to the corresponding histogram interval to populate the histogram.

As all threads work in parallel, we must guarantee a proper synchronization among them, such that only one thread can modify a bin counter at any given time. Synchronization is obtained with OpenCL (or CUDA) proper functions, such as atomic_add. However, this can become a source of overhead, and locally computing parts of the histogram and then merging results together may be a solution. Obviously, there is a trade-off between concurrency and number of threads and kernels, i.e., between fine-grained and coarse-grained procedures. Anyhow, our results do not show large variations in terms of overall performances once the overall ''race-histogram'' strategy has been implemented. For this reason we prefer the simplest solution in which just one single race is run to compute the entire histogram to the one in which more races are run in parallel.

Once this phase is terminated, each box of the histogram represents an interval and it contains the number of features belonging to that interval. To select the target number N of keypoints, we compute a threshold T_H. This threshold is obtained by multiplying the interval size by the number of buckets considered at that point. Then, we use an accumulator A to count the number of features in each interval, starting from the last element of the histogram. The counting phase ends when $A \geq N$. The threshold T_H is then used as a threshold to select or rule-out each keypoint k based on its matching probability k_{mp}. Notice that in this algorithm linear search is performed only on a subset of the overall number of classes. This makes our algorithm much faster than a standard linear search. Moreover, even if the number of features selected is just an estimate of the desired one, this approximation does not have any impact on accuracy as proved in Section VII.

B. HASTENED FS

Algorithm 5 shows how we re-designed FS within the entire CDVS chain to re-position it before OA [11]. We call this computation scheme hastened FS.

Algorithm 5 Hastening FS Before OA

Require: K_O, Q
Ensure: F_O, T_H, D_O
1: **HastenedFS**
2: $(K_O, H) =$ **MP** (K_O, Q)
3: $T_H =$ **ComputeThreshold** (H)
4: $F_O =$ **OrientationAssignment** (K_O, T_H)
5: $H =$ **UpdateHistogram** (H, F_O)
6: $T_H =$ **ComputeThreshold** (H)
7: $D_O =$ **LocalDescriptor** (F_O, T_H)

Input and output parameters are the same of the deferred version (Algorithm 4). However, in this case we first compute the matching probability (line 2), then we compute the threshold T_H (line 3), and finally we resort to the OrientationAssignment function (line 4). Function MP (line 2) works as in Algorithm 4 but, in this case, it manipulates the keypoint set (K_O) instead of the feature set (F_O). Procedure OrientationAssignment starts from the keypoint K_O selected by ComputeThreashold and it produces enriched features F_O, i.e., the original keypoint with added orientation information k_α. Notice that the cardinality of F_O is usually larger than the one of K_O, because OrientationAssignment can generate many features with different orientations from the same keypoint. As a consequence, to maintain the number of selected features as desired, we need to update the histogram (function UpdateHistogram, line 5) and compute a new threshold T_H (line 6). Procedure ComputeThreshold applies the bucket-sort-inspired approximated sorting algorithm previously described in Section V-A. Moreover, all local descriptors D_O are computed by function LocalDescriptor. Notice that in this scheme, functions OrientationAssignment and LocalDescriptor manipulate only a subset of keypoints and features, thus reducing the overall computational effort.

Our experimental analysis, performed on a vast set of images, shows that the contribution of the OA phase to the MP function consists in an almost constant displacement. This means that if we do not consider orientation during MP, no important feature is ruled-out by the process, and the final matching accuracy does not decrease. A similar consideration is done by Lee *et al.* [11]. As a consequence, FS can be

performed before OA computing the orientation only for a reduced number of selected keypoints.

C. MATCHING PROBABILITY (MP)

As analyzed in Sections V-A and V-B the matching probability phase can be applied either after (Algorithm 4) or before (Algorithm 5) orientation assignment. In the first case, it manipulates features $f \in F$. In the second one, it deals with keypoints $k \in K$. For the sake of simplicity, we explicitly refer to Algorithm 5 in this section, and function **MP** will manipulate keypoints.

Following the CDVS standard (see Section III-B), the matching probability phase is mainly based on data evaluated during the detection phase, such as the displacement from the image center (Δ_x, Δ_y), the scale δ, the peak ρ, the curv-sigma σ_{curv}, and the ratio τ. Anyhow, in the deferred algorithm, it also considers data coming from OA, such as α (see Lee *et al.* [33]). **MP** generates the matching probability k_{mp} of each keypoint $k \in K_O$.

To compute these k_{mp} values, the standard adopts several conditional distributions, learned during standardization using an independent matching data set, and several quantification steps [19]. To parallelize and to accelerate their computation (and also to select the most promising features as described in Sections V-A and V-B) we use OpenCL kernel work-groups. A work-group must consist of at least one work-item (thread). 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.

To efficiently distribute the workload of the kernel we use the MP procedure described in Algorithm 6.

Algorithm 6 Computing the Matching Probability k_{mp} for Each Keypoint $k \in K_O$ Q Is a Quantization Set Which Enables the Quantization of the Interest Point Distance From the Image Center, Its Scale, Its Peak Response, Its Curv-Sigma, and Its Ratio

Require: K_O, Q
Ensure: K_O, H

1: **MP**
2: **for** $o = 1$ to $|O|$ **do**
3: **for** $k = 1$ to K_o **do**
4: $k_{\Delta_x,\Delta_y} =$ **Quantize** $(Q, k_{\Delta_x,\Delta_y})$
5: $k_\delta =$ **Quantize** (Q, k_δ)
6: $k_\rho =$ **Quantize** (Q, k_ρ)
7: $k_{\sigma_{curv}} =$ **Quantize** $(Q, k_{\sigma_{curv}})$
8: $k_\tau =$ **Quantize** (Q, k_τ)
9: $(k_{mp}, H) =$ **PDF** $(k_\Delta, k_\delta, k_\rho, k_\sigma, k_\tau)$
10: **end for**
11: **end for**

Function **MP** receives the set of keypoints at all octaves K_O, and a quantification set Q. It returns the same keypoint set K_O updated with new attributes values (namely k_{mp}), and a histogram H. Function **MP** calls function **Quantize** for each keypoint $k \in K_o$ (line 3) of each octave $o \in O$ (line 2). This function restricts keypoint attributes (i.e., the displacement from the image center (Δ_x, Δ_y), the scale δ, the peak ρ, the curv-sigma σ_{curv}, and the ratio τ) to discrete values starting from a continuous set of values. The 5 calls to function **Quantize** (lines 4-8) are done in sequence, but they perform operations with a high level of concurrency. To optimize this phase, we follow the suggestion given by Francini *et al.* [17], where the maximum number n of quantized elements is mapped on the higher populated conditional distributions Q, i.e., $n_{\Delta_x,\Delta_y} = 32$, $n_\delta = 8$, $n_\rho = 16$, $n_{\sigma_{curv}} = 16$, $n_\tau = 16$. Thus, for every **Quantize** function calls, we activate a work group composed by 32 work-items running in parallel. Anyhow, for each function **Quantize** only the right number of threads n (i.e., n_{Δ_x,Δ_y}, n_δ, n_ρ, $n_{\sigma_{curv}}$, and n_τ) is activated. The attribute values of each keypoint k (i.e., k_{Δ_x,Δ_y}, k_δ, k_ρ, $k_{\sigma_{curv}}$, and k_τ) after quantization (computed by lines 4-8) are used by function **PDF** to generate the matching probability k_{mp} of the keypoint k. Finally, one work-item inserts each keypoint matching probability k_{mp} into the histogram H used by function **ComputeThreshold** (see Section V-A) to select the desired number of keypoints.

VI. ORIENTATION ASSIGNMENT (OA)

OA finds the feature $f \in F_O$ corresponding to each keypoint $k \in K_O$. During the orientation assignment phase, CDVS assigns a dominant orientation F_α to each feature to allow rotation invariance during the pairwise matching phase. To do that, for each keypoint selected by the FS (hastened or deferred) phase, CDVS computes the magnitude Θ_{mod} and direction Θ_θ of the gradient of every pixel within the circular neighboring area defined by the keypoint (scale k_δ and sigma k_σ) and from the Gaussian (G_o^f with $f = K_\delta \in (1 \dots G_g)$ and $o = K_o$). Gradient values Θ_θ are then organized in a histogram H_k.

All previous computations are sketched in Fig. 6 and detailed by Algorithm 7. For each keypoint $k \in K$ (black dot) its scale defines a circular neighboring area (or radius r_k and evaluated by function **Compute**, line 4) and a square matrix circumscribing this area (of size equal to $[(2 \cdot r_k) \times (2 \cdot r_k)]$ pixels). The **Gradient** is then used to get the final orientation result [34] adopting a histogram composed by 36 bins (function **Histo**, line 8). This histogram is recursively

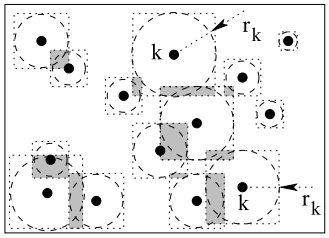


FIGURE 6. Keypoints $k \in K$ (black dots), circular neighboring (of radius r_k), and circumscribed square (of size $[(2 \cdot r_k) \times (2 \cdot r_k)]$ pixels) defined by the keypoint scale.

Algorithm 7 Sequential Algorithm: Pixel-by-Pixel Manipulation for Gradient Computation and Histogram Update. If There Values Are Larger Than 80% of the Main Maximum, the Procedure Duplicates the Keypoint k Into a New Feature

Require: K_O
Ensure: F_O
1: **OrientationAssignment**
2: **for** $o = 1$ to $|O|$ **do**
3: **for** $k \in K_o$ **do**
4: $r_k =$ **Compute** (k_σ)
5: **for** $i = (k_{\Delta_x} - r_k)$ to $(k_{\Delta_x} + r_k)$ **do**
6: **for** $j = (k_{\Delta_y} - r_k)$ to $(k_{\Delta_y} + r_k)$ **do**
7: $(\Theta_{mod}, \Theta_\theta) =$ **Gradient** $(i, j, k_\delta, G_{o,s})$
8: $H_k =$ **Histo** (Θ_θ)
9: **end for**
10: **for** $i = 1$ to 6 **do**
11: $H_k =$ **Smooth** (H_k)
12: **end for**
13: **end for**
14: $k_\alpha =$ **Max** (H_k)
15: **end for**
16: **end for**

smoothed (procedure Smooth, line 11) six times by averaging 3 adjacent bins at a time. The bin corresponding to the highest peak (function Max, line 14), as well as the bins with a value larger than 80% of the highest value, are selected as the dominant orientations of the interest point. Notice that these extra orientations generate the same features computed by the original keypoints but with different orientations.

The OpenCL computation scheme for Θ^h and Θ^v is very close to the Laplacian one, shown in Fig. 4. Input image pixels (left-hand side of Fig. 4) are used to compute the horizontal Θ^h and vertical Θ^v components of the gradient, separately.

For each $G_{o,s}$ every gray scale pixel (x, y, z, w) gradients Θ^h and Θ^v are computed as follows:

$$\begin{aligned}\Theta^h.x &= 0.5 \cdot (G_{o,s}[1].y - G_{o,s}[0].w)\\ \Theta^h.y &= 0.5 \cdot (G_{o,s}[1].z - G_{o,s}[1].x)\\ \Theta^h.z &= 0.5 \cdot (G_{o,s}[1].w - G_{o,s}[1].y)\\ \Theta^h.w &= 0.5 \cdot (G_{o,s}[2].x - G_{o,s}[1].z)\end{aligned} \tag{2}$$

for the horizontal position, and

$$\begin{aligned}\Theta^v.x &= 0.5 \cdot (G_{o,s}[4].x - G_{o,s}[3].x)\\ \Theta^v.y &= 0.5 \cdot (G_{o,s}[4].y - G_{o,s}[3].y)\\ \Theta^v.z &= 0.5 \cdot (G_{o,s}[4].z - G_{o,s}[3].z)\\ \Theta^v.w &= 0.5 \cdot (G_{o,s}[4].w - G_{o,s}[3].w)\end{aligned} \tag{3}$$

for the vertical one. From Θ^h and Θ^v, it is possible to obtain:

$$\begin{aligned}\Theta_{mod} &= \sqrt{\Theta^{h2} + \Theta^{v2}}\\ \Theta_\theta &= \arctan \frac{\Theta^v}{\Theta^h}\end{aligned} \tag{4}$$

To implement Algorithm 7 concurrently, we experimented with two different solutions. In our first implementation, we run a fixed number of threads (corresponding to the maximum required ones) for each neighboring areas, and we eventually disabled the ones that were not required. In the second one we use 4 kernels to orchestrate the overall work-flow, i.e., a first kernel generates all pagination data and then all other kernels manipulate pixels. Experimentally, we discover that the time spent by the pagination kernel is more than balanced by all following "denser" kernels, and that this last solution is about 40% faster than the previous one on average. As consequence, we concentrate on it. It adopts kernels Gradient, Pagination, Orientation, and Peak. They are called in sequence as described by Algorithm 8. These 4 kernels are described in Sections VI-A, VI-B, VI-C, and VI-D.

Algorithm 8 Concurrent Algorithm: Kernels **Gradient**, **Pagination**, **Orientation**, and **Peak** Are Run Sequentially to Generate Features From Keypoints

Require: K_O
Ensure: F_O
1: **OrientationAssignment**
2: $(\Theta_{mod}, \Theta_\theta) =$ **Gradient** $(k_\delta, G_{o,s})$
3: $PT =$ **Pagination** (K_O)
4: $H_k =$ **Orientation** $(PT, id_{width}, id_{pages})$
5: $K_\alpha =$ **Peak** (H_K)

A. KERNEL GRADIENT: GRADIENT EVALUATION

We use the first kernel Gradient (running a matrix of $[\frac{width}{4} \times height]$ threads) to pre-compute the gradient $(\Theta_{mod}, \Theta_\theta)$ of all pixels within the image. More specifically, like in Section IV, we run one thread for each image texel of the convolution texture. Each thread essentially computes the gradient for that coded pixel using Equations (2) and (3). After that, using Equations 4, we generate the texture pairs including Θ_{mod} and Θ_θ for each convolution $G_{o,s}$ (with $s \in S$ and octave $o \in O$). Notice that this sort of pre-computation implies that the gradient is evaluated also for pixels outside all neighboring areas $(r_k, \forall k \in K_O)$. At the same time, it also implies that we compute a single gradient for all pixels belonging to more than one neighboring area (as represented by all gray areas in Fig. 6). As many pixels actually do belong to more than one neighboring area, these two effects generally balance each other. Once we have computed r_k for each keypoint, to perform the two nested cycles of Algorithm 7 in a concurrent way, we run a group of threads (items) for each keypoint. Each item manipulates a pixel within the keypoint neighboring area and it contributes to the generation of the histogram H_k. If we suppose to organize these items with a proper data grid structure, we should use a 2D matrix storing keypoints along rows and items (for each neighboring pixel) along columns.

B. KERNEL PAGINATION: DENSE PAGINATION TABLE

The pagination kernel does not perform any specific CDVS step as it essentially builds a "pagination table" to make the orientation kernel (analyzed in Section VI-C) more efficient. In other words, the pagination kernel pre-computes information items that are common to all threads run by the orientation kernel to improve its performances and to avoid re-computations. The problem and the core idea are the following ones.

The orientation kernel must analyze all pixels of all keypoint neighboring areas (of all octaves) to update a histogram storing keypoint characteristics. To perform this task in a concurrent way, one possible solution consists in running one thread for each pixel of each keypoint. As the number of pixels in the neighboring area varies with the size of this area, one first possibility is to run a number of threads equal to the number of keypoints multiplied by the number of pixels in the largest neighboring area. In this case, many threads would be run and then stopped when dealing with keypoints with smaller neighboring areas, and this would imply that a lot of threads would not be SIMT compliant, reducing the overall efficiency.

To solve this problem we first compute the total number of pixels included in the neighboring areas of all keypoints. Then, as the generated threads have to be distributed over a dense grid to optimize their computation, we create a "pagination table" to store every thread-to-pixel correspondence. Finally, we run a number of threads equal to the number of those pixels and we use the pagination table to allow each thread to manipulate the right pixel within the proper keypoint neighboring area. Whereas this last step is performed by the orientation kernel (see in Section VI-C) all previous ones are performed by the pagination kernel. Its implementation is reported in Algorithm 9.

Algorithm 9 Pagination Process The Scheduled Items Within the Pagination Table Represent a Dense Grid of Parallel Processes

Require: K_O
Ensure: PT
1: **Pagination**
2: $r_k =$ **Compute** (k_σ, k_{mp}, T_H)
3: **if** $r_k > 0$ **then**
4: $(id_{start}, id_{end}) =$ **Page** (C_{atomic}, r_k)
5: $PT =$ **Index** $(id_{start}, id_{end}, k)$
6: **end if**

Kernel Pagination receives the keypoint set K_O as a parameter, and it returns the pagination table PT. It runs one thread for each keypoint $k \in K_O$ of that octave. Each thread executes function Compute (line 2) to compute the neighboring regions radius r_k. Thus the neighboring area will be defined by the square matrix of size $[(2 \cdot r_k) \times (2 \cdot r_k)]$. The value returned by function Compute is larger than zero only for the selected keypoints, i.e., for those keypoints above the threshold T_H which have been selected as features k_{mp}. For those features (line 3), function Page (line 4) assigns to each keypoint a number of threads equal to the number of pixels in its neighboring area. The global counter C_{atomic} is used to count the number of required threads required for each keypoint as well as the total number of threads required by all keypoints. As function Page is shared among all threads, we need to avoid collisions on the counter C_{atomic}. To obtain a proper synchronization we protect the counter within a critical section, such as the one obtained with the OpenCL function atomic_add. As a result, function Page finally returns for each keypoint k a starting index (id_{start}) and an ending index (id_{end}). All threads assigned to that keypoint will have index within the range $[id_{start}, id_{end}]$. As variable C_{atomic} is manipulated within a critical section, its final value defines the total number of threads that have to be run. Those threads are rearranged in groups, spread-out in subsequent pages (i.e., rows) within the pagination table PT. Each page of the pagination table represents a group of constant size, equal to $PT_{width} = 512$. We selected this value statically for the following reasons. First of all, 512 is a power of 2 and this enables each thread to address the right page with a simple binary-shift operation. Obviously, any power of 2 would suit this requirement, but too small values would imply too many rows to represent threads for the same keypoint, and too large values would imply too many keypoints into the same row. In fact, the neighboring area of each feature can be spread in more than one page, and a page may store neighboring items belonging to more than one feature. Groups of items belonging to different pages will have duplicated entries on different pages within PT. Given the smallest neighboring area with pages of width 512, we experimentally noticed that each page contains information for a maximum of 8 keypoint neighboring areas. This is an important value as within the orientation kernel each thread will have to visit an entire page of the pagination table PT to recover its pixel position. Too large values would imply an inefficient sequential search. Too small values would imply a too large number of pages. Thus 512 (corresponding to 8 keypoints) is a trade-off among several requirements.

Function Index finally generates the entry for the pagination table PT based on the item position (id_{start}, id_{end}). PT will be used by all threads within the orientation kernel to recover all required keypoint information. Notice that within the pagination table, each entry is encoded with two pixels represented as RGBA float numbers.

C. KERNEL ORIENTATION: UPDATE HISTOGRAM

Once the pagination table is created we run the Orientation kernel to update all histogram's bins. Once again, this step corresponds to a real CDVS phase, whereas the pagination thread has just set the pagination table up to optimize it. In other words, the orientation kernel uses the pagination table to recover all thread-to-pixel correspondences. Its pseudo-code is reported in Algorithm 10.

It receives as input parameters the pagination table PT and the thread identifiers id_{width} and id_{pages}. It returns the

Algorithm 10 Orientation Kernel: Starting From the Pagination Table (PT) Pre-Computed by the Pagination Kernel, It Creates the Histogram H_k

Require: PT, id_{width}, id_{pages}
Ensure: H_k
1: **Orientation**
2: $(id_k, id_{item}) =$ **Recovery** $(PT, id_{width}, id_{pages})$
3: $H_k =$ **Histo** $(id_k, id_{item}, \Theta_{mod}, \Theta_\theta)$

final keypoint histogram H_k. The orientation kernel runs a number of threads equal to the total number of pixels within all selected keypoints (i.e., features), which also corresponds to the final value of variable C_{atomic}. Each thread, using procedure **Recovery**, accesses the pagination table texture PT (line 2). This function, given the thread identifiers (id_{width} and id_{pages}), identifies the corresponding keypoint id_k and the target pixel id_{item} of which it will be in charge of.

Function **Histo** builds the keypoint histogram. To store the histogram H_k, we use the global memory (shared by all threads run by the **Orientation** kernel) to enable atomic operations. Essentially, the histogram is composed by a buffer with $[2048 \times 36]$ integer elements, where 2048 is the maximum number of keypoints per octave, and 36 is the length of the orientation histogram of each keypoint. The current gradient, is selected using the feature scale k_δ. Each thread reads the direction Θ_θ and magnitude Θ_{mod} of the gradient of the target pixel, it identifies the corresponding orientation histogram bin, and, using atomic instructions (such as the OpenCL **atomic_add**) it adds the corresponding gradient magnitude (multiplied by a specific function [19]) to the histogram.

D. KERNEL PEAK: SMOOTH HISTOGRAM

The fourth and last kernel performs the smoothing and it selects the dominant orientations. This kernel runs $[K_O \times 36]$ threads. Its pseudo-code is reported in Algorithm 11.

Algorithm 11 Kernel **Peak** Returns All Keypoint Orientations (K_α) Whose Values Is Larger Than 80% of the Maximum Orientation Value

Require: H_k
Ensure: K_α
1: **Peak**
2: **for** $i = 1$ to 6 **do**
3: $H_k =$ **Smooth** (H_k)
4: **end for**
5: $K_\alpha =$ **Race** (H_k)

We use OpenCL work-groups, composed by 36 work-items, to smooth the orientation histogram H_k. The structure named H_k is used to define the orientation k_α and identify all possible keypoint duplicates. Each work-item operates on an orientation histogram bin computing average values using two adjacent bins. Work-items are synchronized using the OpenCL barrier function in order to read consistent memory values. Following the standard, function **Smooth** is called 6 times. Through loop unrolling and stencil code optimization it performs the required smoothing. Once all computations on all bins have been completed by all work-items, the entire histogram H_k is manipulated by the first 4 work-items within function **Race**. At this point, each work-item computes the maximum histogram value of 9 consecutive histogram bins, e.g., the first work-item computes the maximum among histogram elements in position 1–9, the second one the maximum among histogram elements in position 10–18, etc. After that, the first work-item evaluates the global maximum considering the 4 local maximum values previously computed. Finally each work-item checks whether its corresponding bin is equal to the maximum value, or larger than 80% of the maximum. The maximum bin will define the orientation of the corresponding feature. All other relative maximum values will duplicate the feature with its orientation. All these features will be returned as K_α.

VII. LOCAL DESCRIPTOR COMPUTATION (LDC)

Local descriptor computation is quite similar to the orientation assignment phase, analyzed in Section VI. LDC is performed by 3 kernels.

The first kernel is in charge of all initial operations necessary to compact all items within the pagination table. It follows an implementation quite similar to the one reported in Algorithm 9 of Section VI-B. Anyhow, it is worthwhile to highlight two main differences from Section VI-B:

- The neighboring area is constructed (see function **Compute**) with a different strategy and r_k is usually two times larger than in the pagination kernel. This obviously implies larger computational costs.
- Procedures **Histo** and **Smooth** of Sections VI-C and VI-D are replaced by functions **Descr** and **Norm**. These functions, reported in Algorithms 12 and 13, perform a sequence of steps, described within the standard, to compute the feature descriptor. Those operations are highly parallelizable with subsequent important efficiency advantages.

In the second kernel, reported in Algorithm 12, the **Recovery** phase (line 2) identifies the feature and its corresponding position in the neighboring area.

Algorithm 12 Local Descriptor Computation Kernel

Require: PT, id_{width}, id_{pages}
Ensure: D_f
1: **Descriptor**
2: $(id_k, id_{item}) =$ **Recovery** $(PT, id_{width}, id_{pages}, P_{recovery})$
3: $D_f =$ **Descr** $(id_f, id_{item}, \Theta_{mod}, \Theta_\theta)$

In function **Descr** (line 3), each item participates to the composition of D_f, which is composed by 128 bins. The descriptor D_f is built using data such as Θ_{mod} and Θ_θ stored in textures previously created. D_f includes information used

by the subsequent CDVS matching phase. Differently from Histo, within function Descr each bin is coupled with more than one field of the descriptor D_f. Anyway, also in this case all descriptors are recorded into the global memory to enable atomic operations (e.g., such as the ones performed by the atomic_add function) to avoid conflicts among different items.

The third kernel (running $[K_o \times 128]$ threads) is described in Algorithm 13.

Algorithm 13 Normalization Kernel

Require: D_f
Ensure: D_f
1: **Normalization**
2: **for** $i = 1$ to 2 **do**
3: $D_f =$ **Norm** (D_f)
4: **end for**

It performs the normalization step required by the standard in a concurrent way. The required operations are performed on D_f by groups of 128 threads appropriately synchronized. Following the CDVS standard, normalization is performed twice (line 2). Function Norm (line 3) copies each descriptor from the global memory to a local texture. Loop unrolling is used to optimize stencil codes within the function. The texture has 32 rows and F_O columns. Each descriptor is encoded on 32 RGBA floats in a texture where on each row there is a different D_f.

A. COMPRESSION AND COORDINATE CODING

Local descriptors D_f require about 83 KBytes of memory. This makes local descriptors complex to manage and to transmit. This is especially true when hardware platforms have limited resources, like embedded devices and cell phones. For this reason MPEG developed a compression technique to generate more manageable data. CDVS includes 6 modes of compression that store the following amount of information: 512Bytes, 1KBytes, 2KBytes, 4KBytes, 8KBytes, and 16KBytes. According to the compression mode being used, the compression algorithm selects a specific subset of the transformed components from local descriptors. In order to enable mode interoperability, these components are selected such that the set of components of a more compressed mode is always a subset of the set of components of a less compressed mode. Once the transformed components have been selected, each component is quantized to three values (that is, -1, 0, and 1) and finally encoded (into 10, 0, and 11). The quantization levels for each component are defined in the standard's normative look-up table. These values are obtained maintaining a high level of precision when doing image matching.

For each compression mode, we implement a different kernel. The system runs only the kernel corresponding to the selected mode. Each kernel gathers all coding operations required by an increasing number of groups, depending on the mode (5, 5, 10, 16, 20, and 32). Each group of 8 threads applies the CDVS standard encoding and quantization table to the descriptor to reduce its size as specified by the selected compression mode. Using separate kernels to perform compression enables a straightforward mapping of several coding tables, it reduces the required checks, and improves performances.

The last step in the feature compression pipeline efficiently encodes the coordinates of each keypoint (Δ_x, Δ_y). Coordinates are usually represented using floating-point precision, which becomes a bottleneck once the feature vectors have been quantized. CDVS uses a location histogram coding scheme [35] to identify cluster of features and it efficiently makes use of arithmetic encoding. For this reason, our last kernel $[F_O]$ reads keypoint positions and, based on coding tables, it associates these keypoints to an encoding coordinate which will be used (together with the compact descriptor) during the matching phase.

VIII. EXPERIMENTAL RESULTS

In this section we present our results, with specific attention on accuracy (precision) and computation efficiency (speed). We follow the inter-operability (Pairwise Matching) test [36] defined by the CDVS standard. The reference picture descriptors for pairwise matching are extracted using the Test Model 14 (TM 14.0) (the last implementation of the standard at publication time) while the query descriptors are selected using the proposed methods. The data-sets used for the CDVS interoperability test are the followings [37]: Graphics (2500 images), Graphics VGA resolution (2500 images), Graphics VGA resolution and high JPEG compression (2500 images), Paintings (455 images), Video Frames (500 images), Buildings (14935 images), Common Objects (10200 images). We manipulate all images with a resolution of $(640 \cdot 480)$ pixels.

We selected three widely used devices:

- The Arndale Octa board. This is a high-power single-board computer featuring a 1.7GHz dual-core ARM Cortex-A15 as a CPU and an ARM Mali T628 GPU.
- The Samsung Galaxy Note 3. This adopts a chipset Snapdragon 800 Qualcommm MSM8974 embedding a CPU 2.3GHz quad-core ARM Cortex A15 and a Qualcomm Adreno 330 GPU.
- The Samsung Galaxy Note 4. It uses a APQ8084 Qualcomm Snapdragon 805 chipset, with a 2.7 GHz Quad Core CPU and a Qualcomm Adreno 420 GPU.

Anyhow, our considerations may easily be extended to other float single-precision GPUs supporting the Open-CL or CUDA language.

Section VIII-A compares our fully GPU-based approach with the CPU-based CDVS Test Model, in terms of time efficiency. Section VIII-B draws a comparison between our approach and the one by Duan *et al.* [16]. Moreover, following Cavicchioli *et al.* [38], it talks about memory issues and memory transfer times. Section VIII-C, follows Section VIII-A, and it compares the same approaches in terms of accuracy.

A. EFFICIENCY

We start our experimental analysis by comparing our GPU-based implementation of the extraction pipeline of Fig. 1 with the CPU-based CDVS Test Model implementation.

Fig. 7 reports results on the whole CDVS data-set,i.e., more than 33, 000 images with resolution $(640 \cdot 480)$ pixels. The y-axis indicates the average feature extraction times, i.e., the average wall-clock times,[2] for 4 different approaches (reference deferred computation and hastened computation on the CPU, and deferred and hastened computations on the GPU) running on the three selected target devices (i.e., the Arndale Octa board, Fig. 7a, the Samsung Galaxy Note 3, Fig. 7b, and the Samsung Galaxy Note 4. Fig. 7c). The x-axis reports the different CDVS descriptor lengths measured in bytes (512, 1K, 2K, 4K, 8K, 16K). Average values are computed over all images. The plots show that the reference implementation (named "Reference Deferred") is from 2 to 5 times slower than our OpenCL implementations. This is true especially for the larger descriptor lengths, as GPUs scale much better than CPUs when working with larger amount of data due to efficient massive parallelization. Notice that the two CDVS reference implementations on the Arndale and the Note 3 platforms have very similar results, as they embed the same CPU. Moreover, "hastened" versions are from 10% to 30% faster than "deferred" versions, and they use less memory, reducing interference issues on common memory platforms. This difference is larger for GPU platforms than for CPU ones.

Fig. 8 presents time breakdowns of our techniques and comparison with the reference implementation on the Samsung Galaxy Note 3. Similar results have been collected on the other platforms. The x-axis reports the descriptor lengths (in bytes), for the reference CPU implementations (deferred and hastened) and for the GPU versions (deferred and hastened). The y-axis reports average wall-clock times of the main phases (KD, OA, FS, LDC). For the hastened implementation the FS stage has been divided into FS1 and FS2 (to indicate the time spent before and after, respectively, the OA phase). The average height of the CDVS reference

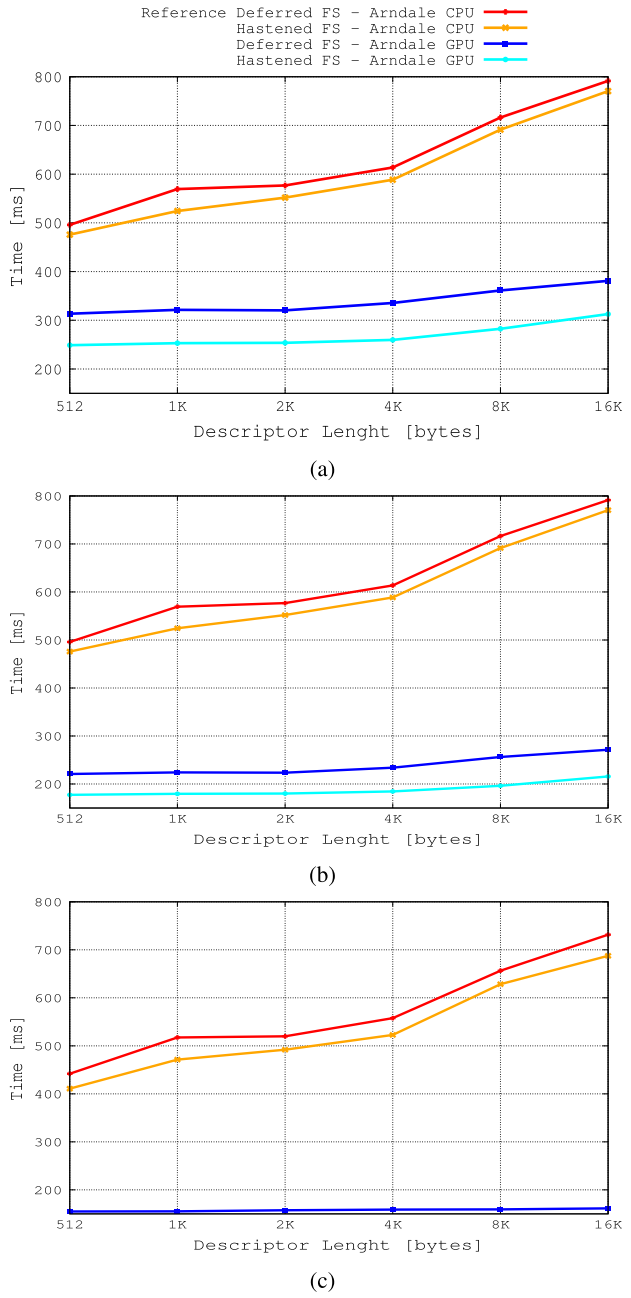


FIGURE 7. Average wall-clock running times on the entire CDVS data-set. Times include all CDVS processing steps, from the original input images to the extraction of compact descriptors (see Fig. 1). (a) Arndale Octa board. (b) Samsung galaxy note 3. (c) Samsung galaxy note 4.

[2]The wall-clock time is the time necessary to a (mono-thread or multi-thread) process to complete its job on a new input image, i.e., the difference between the time at which an image is completely handled and the time at which this task started. For this reason, the wall-clock time is also known as "elapsed time".

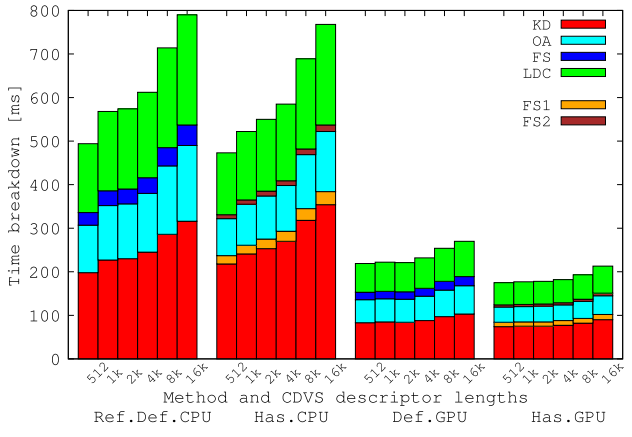
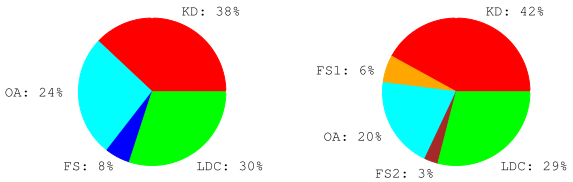


FIGURE 8. Average running times for the Samsung Galaxy Note 3 with time breakdown in the different algorithmic phases. All times are reported in milliseconds. The 4 histograms report (from left to right) time details for the: (a) Reference (deferred) CPU algorithm, (b) Hastened CPU version, (c) Deferred GPU implementation, (d) Hastened GPU version, All 4 histograms report times for all CDVS descriptor lengths.

CPU bars are about 627 (deferred) and 600 (hastened) milliseconds, whereas the average height of the GPU runs are 238 (deferred) and 189 (hastened).

Fig. 9 presents time breakdowns of our GPU implementations (deferred and hastened versions) on a pie chart. The KD stage is strongly influenced by the image size and it is the most expensive step. Nevertheless, it can be highly optimized by the concurrent implementation. It approximately needs the 40% of the total time on the GPU on average, whereas this figure is higher for the CPU implementation. The OA and FS stages require much less time than the KD stage, but their concurrent versions are not much faster than the sequential ones.



(a) (b)

FIGURE 9. GPU time breakdown: Deferred approach (4 main phases, i.e., KD, FS, OA, LDC) and hastened version (5 main phases, i.e., KD, FS1, OA, FS2, LDC). (a) Deferred FS. (b) Hastened FS.

Table 1 reports the resulting average speed-up for the GPU with respect to the CPU. The most expensive phases have an average speed-up larger than 3 for thee hastened approach, and slightly smaller than 3 for the deferred algorithm. Our concurrent implementation makes smaller improvements to the FS and the OA phases whose speed-up is around 2.

TABLE 1. GPU versus CPU Speed-up: Deferred approach (4 main phases, i.e., KD, FS, OA, LDC) and Hastened version (5 main phases, i.e., KD, FS1, OA, FS2, LDC).

Deferred Approach			
KD	FS	OA	LDC
2.77	2.41	1.97	2.81

Hastened Approach				
KD	FS1	FS2	OA	LDC
3.48	2.12	2.86	2.12	3.29

B. COMPARISON AND MEMORY ISSUES

In this section we compare our results with the ones by Duan *et al.* [16], and we present some data to justify our idea to avoid CPU-to-GPU memory transfer as long as possible and to use the integrated memory. For this reason, in this section, we mainly concentrate on running times, throughput, and latency.

Table 2 reports a comparison with the data presented by Duan *et al.* [16].

In that paper the authors present a CDVS implementation targeting discrete platforms with cooperating CPU and GPU cores. To speed-up the process, the authors also incorporate several optimizations within the standard CDVS encoder, such as the adoption of deep learning based approaches.

TABLE 2. Comparison between our approach and data presented by Duan *et al.* [16]. The symbol – means that the data is not available on the original paper.

CPU/GPU		Time [ms] [16]	Time [ms] This paper
CPU	Snapdragon 800	–	576.5
	Snapdragon 805	–	514.3
	Intel Xeon E5-2650	116.7	154.8
GPU	Adreno 330 GPU	–	158.7
	Adreno 420 GPU	–	101.5
	NVIDIA GTX 1060	5.1	7.9

On the contrary, we mainly concentrate on embedded systems with much more limited computing power and our target is not to optimize single CDVS phases but to stick to the standard as much as possible while porting it under a many-core architecture. Moreover, one of our algorithmic intrinsic features is to avoid memory transfer to possibly let the CPU work on other issues. Albeit these very different starting points, we run some experiments to have some common figures. First of all, we select two common hardware architectures. Then we show the average running time to extract different visual descriptors on 1000 images with resolution $(640 \cdot 480)$ pixels. We report results on our reference CPUs (Snapdragon 800 and Snapdragon 805), GPUs (Adreno 330 and Adreno 420), and on the common platforms (i.e., Intel Xeon E5-2650 v2 2.6GHz CPU and NVIDIA GTX 1060 GPU). As there are no details on how the images were selected, we present average results on 10 different sets of 1000 images. Table 2 reports our data.

Although, as we have just mentioned, the two strategies are quite different, and no definitive conclusion can be made, Table 2 at least shows that our results and the ones by Duan *et al.* [16] are somehow comparable. We need more time to run experiments, but this is not surprising, and from our point of view, it can be considered as a very good result given the consideration reported above. Moreover, one of the main differences is that Duan *et al.* [16] mainly concentrates on discrete devices. On these platforms memory copies mainly involve copying data from the system DRAM towards the on-board RAM of the graphics adapter, through the PCI standard. On the contrary, we concentrate on embedded systems, where integrated graphics processors may share memory on modern platforms. However, these approaches introduce CPU-to-GPU memory coherency problems when accessing the same shared memory buffer. As a consequence sharing memory and avoiding memory copies does not necessarily lead to performance improvements. To analyze this issue, and following Cavicchioli *et al.* [38], we present an analysis in which we show how the performance of our algorithm degrades on recent SoCs, when memory is shared among the CPU cores and the GPUs (see also the considerations reported at the end of Section III-C).

Fig. 10 reports the following experiments.

We consider an Intel i7-6700 SoC platform, featuring an HD 530 Integrated GPU. This platform uses OpenCL 2.0, and this, in turn, enables shared memory usage. In this platform

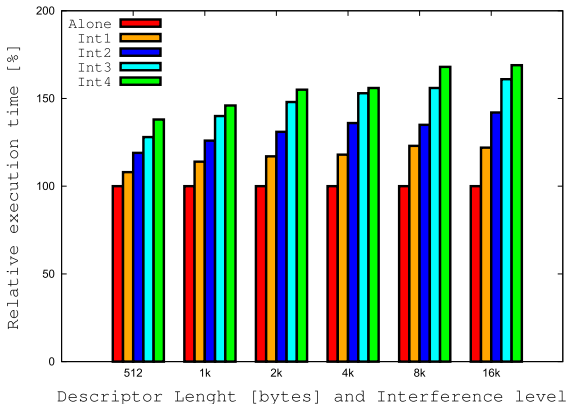


FIGURE 10. Relative execution times with different levels of interference (from no interference at all `Alone`, to the larger one `Int4`) between the CPU and the GPU.

we dedicate the GPUs to run our CDVS algorithm with the different compression modes (512B, 1KBytes, 2KBytes, 4KBytes, 8KByets and 16KBytes), whereas we run some interference programs on the CPU cores. The x-axis reports the different compression modes. For each mode the histogram bars refer to the different interference programs. Interference programs perform sequential and random accesses, in read and write modes, involving an increasing working memory size and an increasing number of CPU cores. The y-axis indicates the total relative execution time averaged on all experiments. Data shows that slow-down of more than 60% may be reported in some cases.

C. ACCURACY

Accuracy is a primary target for the CDVS standard and a main issue when working with GPU [39]. To evaluate the accuracy of our algorithms, we present the CDVS inter-operability test results. Accuracy results are practically identical on all platforms. Thus, we just report results for the Samsung Note 3. We work with 33, 000 images, running the inter-operability test on the entire CDVS data-set. We generate about 17, 000 matching-pairs and about 180, 000 no-matching-pairs.

Fig. 11 shows the true positive rate for the 4 different approaches previously analyzed.

The x-axis reports the query size defined by the CDVS standard. The sizes (1k, 4k) and (2k, 4k) represent the matching performed with different descriptor lengths, e.g., 1k descriptors length are compared with 4k ones. The y-axis reports the true positive rate for the 4 implementations. The high variability of the true positive rate is due to the intrinsic characteristics of the test model. Notice that, the monotonicity of the curve should be obtained by re-ordering the x-axis (as 512, 1K, 1K-4K, 2K, 2K-4K, 4K, 8K, 16K), but we did not perform this operation, as we decided to adopt the order specified in the original MPEG proposal. Anyway, the plot shows that the difference between our implementations and the reference one is less than 2% for the majority of the modes.

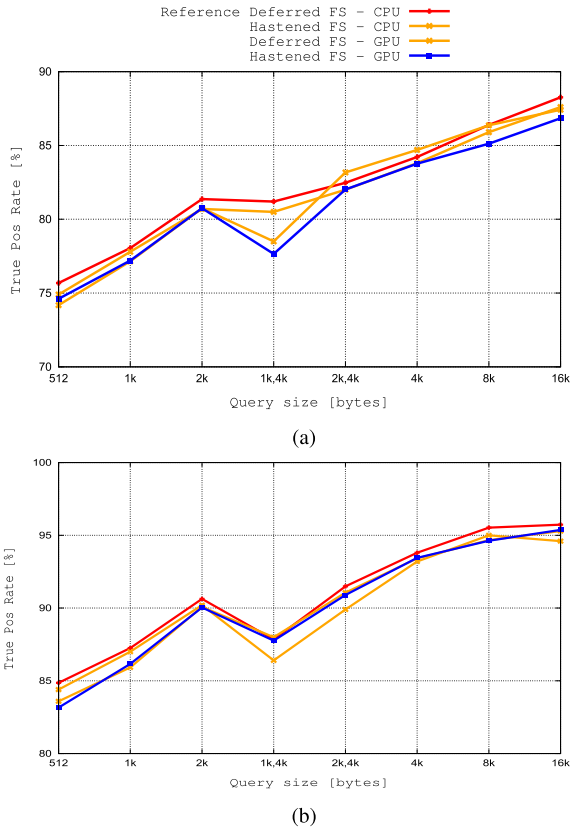


FIGURE 11. CDVS data-set: Pairwise matching true-positive rate. (a) Buildings. (b) Common objects.

Fig. 12 shows the false alarm rate for the same approaches under the same conditions.

In this case, the difference between our implementations and the reference one is less than 0.2%. The two graphs prove that the visual matching systems under test have similar behaviors in terms of accuracy and that the exact response (i.e., the pair is matching or non-matching) is given with a very high probability.

We finally compare our approaches with the reference one considering pairwise matching test results performed using the same approach to extract the features, i.e., conversely to the inter-operability test, described above, the reference and the query image features are computed by the same extractor. The objective of this test is to analyze the intrinsic characteristics, in terms of accuracy, of our approaches considering them as stand-alone visual search systems. To this regard we selected from [40] and [41] data-sets about 2500 images, generating about 10000 matching-pairs and about 5000 non-matching pairs. Fig. 13 shows the Receiver Operating Characteristic (ROC[3]) for the CPU and GPU implementations on the same benchmark set.

Even if we performed the test on all reference GPUs, we report results just on the Samsung Galaxy Note 3 device, as the data are identical in the other cases. Essentially,

[3]A ROC curve is a graphical plot which illustrates the performance of a binary classifier system as its discrimination threshold is varied (please, see reference [42] for further details).

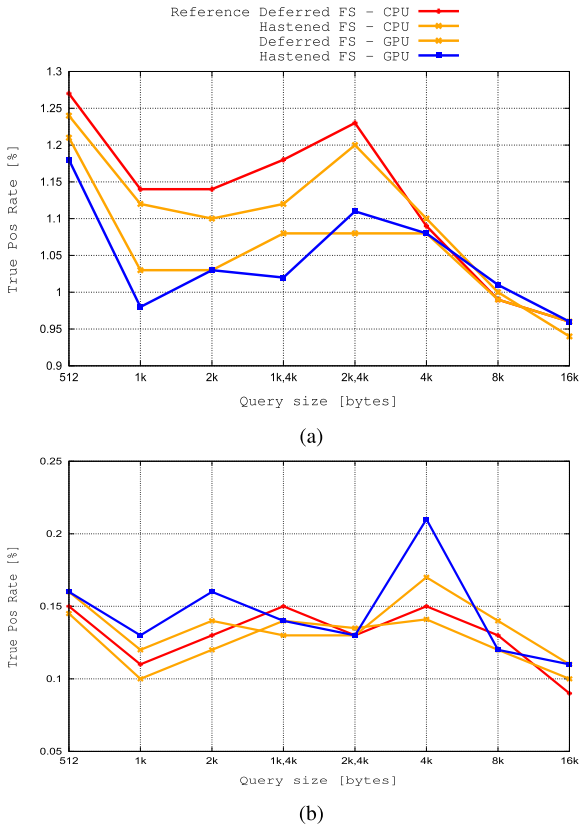


FIGURE 12. CDVS data-set: Pairwise matching false-positive rate. (a) Buildings. (b) Common objects.

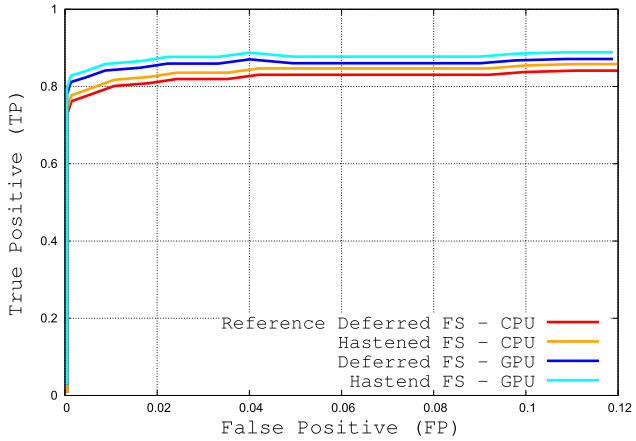


FIGURE 13. GPU and CPU Receiver Operating Characteristics (ROC).

the x axis reports the complemented value of the true negative matching rate ($1-TrueNegative$, i.e., $1-TN$), while the y axis reports the true positive matching rate (TP). 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 0.86. This means that, for a wide range of threshold values, the number of true positives (i.e., correct matching results) remains around 86% (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.

IX. CONCLUSIONS

Pairwise matching has become a core technology for many modern scenarios and several common applications. The extraction of features essential to the matching has received specific attention by several researchers and by the MPEG. Following this standard, extracting features from an image requires several complex and time consuming steps.

In this paper, we show how to efficiently implement the entire process of descriptors extraction, i.e., all main stages of the CDVS standard and the ALP detector, on embedded GPUs. We specifically focus on the more time consuming and complex phases, namely keypoint detection, feature selection, orientation assignment, and local descriptor computation. We discuss strategies and recommendations to divide the overall workload among different kernels. We show how to enforce thread regularity. We strive to improve concurrency and reduce synchronization waiting times. We present how to design "pagination" or "recovery" tables to store and access data efficiently from different threads. We debate how to appropriately store and retrieve all data that have to be transferred from the CPU to the GPU (and vice-versa) and exchanged among different kernels.

In our approach the entire workload and data flow have been maintained within the GPU and GPU only. This approach may lead to some inefficiency, but it also intrinsically increases concurrency, it avoids repeated data transfer between different computing units, and it keeps the CPU idle as long as possible. This in turns enables the CPU to work on other tasks that may be deemed as necessary on embedded and power-limited systems.

Experimental results on CDVS standard image data-sets shows that our solutions have a speed-up up to 3x over the CDVS Test Model CPU implementation. Moreover, pairwise-matching experiments clearly show that our parallel implementations are very close to the test model one in terms of accuracy.

REFERENCES

[1] B. Girod *et al.*, "Mobile visual search," *IEEE Signal Process. Mag.*, vol. 28, no. 4, pp. 61–76, Jul. 2011.
[2] L.-Y. Duan, F. Gao, J. Chen, J. Lin, and T. Huang, "Compact descriptors for mobile visual search and MPEG CDVS standardization," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Beijing, China, May 2013, pp. 885–888.
[3] K. Cordes, B. Rosenhahn, and J. Ostermann, "Localization accuracy of interest point detectors with different scale space representations," in *Proc. IEEE Int. Conf. Adv. Video Signal Based Surveill. (AVSS)*, Seoul, South Korea, Aug. 2014, pp. 247–252.
[4] T. Lindeberg, "Discrete derivative approximations with scale-space properties: A basis for low-level feature extraction," *J. Math. Imag. Vis.*, vol. 3, no. 4, pp. 349–376, Nov. 1993.
[5] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proc. IEEE Int. Conf. Comput. Vis.*, Washington, DC, USA, vol. 2, Sep. 1999, pp. 1150–1157.
[6] G. Francini *et al.*, "Accurate and efficient visual search on embedded systems," in *Proc. 3rd Int. Conf. Adv. Comput., Commun. Inf. Technol. (CCIT)*, Birmingham, U.K., Apr. 2015, pp. 61–66.

[7] A. Garbo, C. Loiacono, S. Quer, M. Balestri, and G. Francini, "CDVS feature selection on embedded systems," in *Proc. IEEE Int. Conf. Multimedia Expo Workshops (ICMEW)*, Turin, Italy, Jun./Jul. 2015, pp. 1–6.
[8] G. Wang, B. Rister, and J. R. Cavallaro, "Workload analysis and efficient OpenCL-based implementation of SIFT algorithm on a smartphone," in *Proc. IEEE Global Conf. Signal Inf. Process. (GlobalSIP)*, Austin, TX, USA, Dec. 2013, pp. 759–762.
[9] M. Suárez, V. M. Brea, J. Fernández-Berni, R. Carmona-Galán, D. Cabello, and A. Rodríguez-Vázquez, "A 26.5 nj/px 2.64 Mpx/s CMOS vision sensor for Gaussian pyramid extraction," in *Proc. 40th Eur. Solid State Circuits Conf. (ESSCIRC)*, Sep. 2014, pp. 311–314.
[10] P. Leyva *et al.*, "Simplification and hardware implementation of the feature descriptor vector calculation in the SIFT algorithm," in *Proc. 24th Int. Conf. Field Program. Logic Appl. (FPL)*, Sep. 2014, pp. 1–4.
[11] K. Lee, S. Lee, and W.-G. Oh, "Accelerating local feature extraction using two stage feature selection and partial gradient computation," in *Proc. Asian Conf. Comput. Vis.*, 2014, pp. 366–380.
[12] S. Zhang, R. Wang, Q. Wang, and W. Wang, "Accelerating CDVS extraction on mobile platform," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Quebec City, QC, Canada, Sep. 2015, pp. 3837–3840.
[13] O. J. Arndt, T. Linde, and H. Blume, "Implementation and analysis of the histograms of oriented gradients algorithm on a heterogeneous multicore CPU/GPU architecture," in *Proc. IEEE Global Conf. Signal Inf. Process. (GlobalSIP)*, Dec. 2015, pp. 1402–1406.
[14] I. A. Doush and S. AL-Btoush, "Currency recognition using a smartphone: Comparison between color SIFT and gray scale SIFT algorithms," *J. King Saud Univ.-Comput. Inf. Sci.*, vol. 29, no. 4, pp. 484–492, Oct. 2017.
[15] C. Lee, C. E. Rhee, and H.-J. Lee, "Complexity reduction by modified scale-space construction in SIFT generation optimized for a mobile GPU," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 27, no. 10, pp. 2246–2259, Oct. 2017.
[16] L.-Y. Duan *et al.*, "Fast MPEG-CDVS encoder with GPU-CPU hybrid computing," *IEEE Trans. Image Process.*, vol. 27, no. 5, pp. 2201–2216, May 2018.
[17] G. Francini, S. Lepsøy, and M. Balestri, "Selection of local features for visual search," *Signal Process., Image Commun.*, vol. 28, no. 4, pp. 311–322, Apr. 2013.
[18] T. Lindeberg, "Feature detection with automatic scale selection," *Int. J. Comput. Vis.*, vol. 30, no. 2, pp. 79–116, 1998.
[19] L.-Y. Duan *et al.*, "Overview of the MPEG-CDVS standard," *IEEE Trans. Image Process.*, vol. 25, no. 1, pp. 179–194, Jan. 2016.
[20] M. Doggett, "Texture caches," *IEEE Micro*, vol. 32, no. 3, pp. 136–141, May 2012.
[21] S. Bianco, D. Mazzini, D. P. Pau, and R. Schettini, "Local detectors and compact descriptors for visual search: A quantitative comparison," *Digit. Signal Process.*, vol. 44, pp. 1–13, Sep. 2015.
[22] A. Witkin, "Scale-space filtering: A new approach to multi-scale description," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process. (ICASSP)*, San Diego, CA, USA, Mar. 1984, pp. 150–153.
[23] S. Chen, J. Qin, Y. Xie, J. Zhao, and P.-A. Heng, "A fast and flexible sorting algorithm with CUDA," in *Proc. 9th Int. Conf. Algorithms Archit. Parallel Process. (ICAPP)*, Taipei, Taiwan, Jun. 2009, pp. 281–290.
[24] M. Lu, B. He, and Q. Luo, "Supporting extended precision on graphics processors," in *Proc. 6th Int. Workshop Data Manage. New Hardw. (DaMoN)*, Indianapolis, IN, USA, Jun. 2010, pp. 19–26.
[25] M. Burtscher and K. Pingali, "An efficient CUDA implementation of the tree-based Barnes hut *n*-body algorithm," in *Proc. GPU Comput. Gems Emerald Ed.*, Burlington, VT, USA, 2011, pp. 75–92.
[26] M. E. Lalami, D. El-Baz, and V. Boyer, "Multi GPU implementation of the simplex algorithm," in *Proc. IEEE Int. Conf. High Perform. Comput. Commun.*, Banff, AB, Canada, Sep. 2011, pp. 179–186.
[27] G. Lowe, "Concurrent depth-first search algorithms based on Tarjan's algorithm," *Int. J. Softw. Tools Technol. Transf.*, vol. 18, no. 2, pp. 129–147, 2016.
[28] C. Johnson, L. Barford, S. M. Dascalu, and F. C. Harris, Jr., "CUDA implementation of computer go game tree search," in *Proc. 13th Int. Conf. Inf. Technol. New Gener.*, Las Vegas, NV, USA, Mar. 2016, pp. 339–350.
[29] Z. S. Hakura and A. Gupta, "The design and analysis of a cache architecture for texture mapping," *ACM SIGARCH Comput. Archit. News*, vol. 25, no. 2, pp. 108–120, 1997.
[30] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying GPU microarchitecture through microbenchmarking," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Mar. 2010, pp. 235–246.
[31] L.-Y. Duan, J. Lin, J. Chen, T. Huang, and W. Gao, "Compact descriptors for visual search," *IEEE Multimedia*, vol. 21, no. 3, pp. 30–40, Jul./Sep. 2014.
[32] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. L. B. Kim, and P. Dubey, "Fast sort on CPUs and GPUS: A case for bandwidth oblivious SIMD sort," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 351–362.
[33] K. Lee, S. Lee, S.-I. Na, S. Je, and W.-G. Oh, "Extensive analysis of feature selection for compact descriptor," in *Proc. 19th Korea-Jpn. Joint Workshop Frontiers Comput. Vis.*, Incheon, South Korea, Jan./Feb. 2013, pp. 53–57.
[34] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vis.*, vol. 60, no. 2, pp. 91–110, 2004.
[35] S. S. Tsai *et al.*, "Improved coding for image feature location information," *Proc. SPIE*, vol. 8499, pp. 8499-1–8499-10, 2012, doi: 10.1117/12.935619.
[36] *MPEG-CDVS Group*. Accessed: Mar. 1, 2015. [Online]. Available: http://wg11.sc29.org
[37] *MPEG-CDVS Group*. Accessed: Mar. 1, 2015. [Online]. Available: http://pacific.tilab.com/www/datasets/download/Dataset-20120210
[38] R. Cavicchioli, N. Capodieci, and M. Bertogna, "Memory interference characterization between CPU cores and integrated GPUS in mixed-criticality platforms," in *Proc. 22nd IEEE Int. Conf. Emerg. Technol. Factory Automat. (ETFA)*, Sep. 2017, pp. 1–10.
[39] G. Cabodi, A. Garbo, C. Loiacono, S. Quer, and G. Francini, "Efficient complex high-precision computations on GPUS without precision loss," *J. Circuits, Syst. Comput.*, vol. 26, no. 12, p. 1750187, 2017.
[40] T. Italia. *The Cturin180 Test Set*. Accessed: Jan. 15, 2016. [Online]. Available: http://jol.telecomitalia.com/jolvisible/cturin180/?lang=en
[41] Computer Vision Laboratory. *Zurich Building Image Database*. Accessed: Oct. 1, 2014. [Online]. Available: http://www.vision.ee.ethz.ch/showroom/zubud/index.en.html
[42] F. Oberti, A. Teschioni, and C. S. Regazzoni, "ROC curves for performance evaluation of video sequences processing systems for surveillance applications," in *Proc. IEEE Int. Conf. Image Process. (ICIP)*, Kobe, Japan, Oct. 1999, pp. 949–953.

ALESSANDRO GARBO received the M.S. degree in software engineering from the Politecnico di Torino, Italy, in 2003, and the Ph.D. degree in software engineering from the Dipartimento di Automatica ed Informatica, Politecnico di Torino, in 2007. He was as a Post-Doctoral Researcher with the Computer Engineering Department, Politecnico di Torino, with Prof. G. Cabodi and Prof. S. Quer for 10 years. He is currently a Senior Researcher with Nuance, where he contributes in the research and development of new algorithms in speech synthesis. His research interests include vision algorithms for automotive applications, eye/gaze tracking, traffic video detection, image processing, and pattern recognition. He is active in the computer vision field using embedded system for tracking and machine learning.

STEFANO QUER received the M.S. degree in electronic engineering from the Politecnico di Torino, Turin, Italy, in 1991, and the Ph.D. degree in computer engineering from the Ministry of University and Scientific and Technological Research, Rome, in 1996. He has been a Visiting Faculty with the Department of Electronic Engineering and Computer Science, University of California at Berkeley. He has been an Intern with the Advanced Technology Group, Synopsys, Inc., Mountain View, CA, USA, and the Alpha Development Group, Compaq Computer Corporation, Shrewsbury, MA, USA. He has been a Compaq Computer Corporation Consultant. He is currently a Professor with the Department of Control and Computer Engineering, Politecnico di Torino. His main research interests include systems and tools for CAD for VLSI, formal methods for hardware and software systems, and embedded systems. Other activities focus on the development of sequential and concurrent algorithms and on optimization techniques able to achieve acceptable solutions with limited resources.