Accelerating SURF Detector on Mobile Devices

Xin Yang and Kwang-Ting (Tim) Cheng
Dept. of Electrical and Computer Engineering, University of California, Santa Barbara, CA, 93106, USA
xinyang@umail.ucsb.edu, timcheng@ece.ucsb.edu

ABSTRACT

Running a SURF (Speeded Up Robust Features) detector on mobile devices remains too slow to support emerging applications such as mobile augmented reality. Porting it without adapting the algorithm to account for mobile platform limitations could result in significant runtime degradation. In this paper, we identify two mismatches between the SURF algorithm and the mobile hardware that cause substantial slow-down of the point detection process: 1) *mismatch between the data access pattern and the small cache size, and 2) mismatch between the huge amount of branches and high pipeline hazard penalty.*

To address the mismatches, we propose two techniques: *tiled SURF* and *gradient moment based orientation assignment*. Tiled SURF improves data locality and greatly reduces memory traffic. A method for determining the optimal tile sizes, named *content-aware tiling*, is designed to minimize runtime and maximize detection accuracy. To avoid the penalties caused by pipeline hazards, we replace the original orientation operator with branching-free gradient moment computations.

The proposed techniques are tested on three mobile platforms. Comparing to the original SURF, the accelerated SURF achieves a 6x~8x speedup without sacrificing recognition accuracy. Meanwhile, it achieves 59%~80% reductions in the runtime ratio of the detector running on mobile platforms compared with on x86-based PCs.

Categories and Subject Descriptors:

I.4.7 [Image Processing and Computer Vision]: Feature Measurement – *feature representation*.

Keywords

SURF, point detection, acceleration, mobile phones, data access patterns, cache miss, branch prediction, pipeline hazards penalty

1. INTRODUCTION

With the pervasive presence of low-cost, high quality cameras on mobile devices such as smartphones, we are witnessing an explosive growth of embedded computer vision applications like mobile augmented reality, mobile object recognition and image search. Interest point detection is a common task for these applications. Over the past decades, several detectors [1, 2, 3, 4, 5, 16] have been developed. The SURF detector [1], which stands for Speeded Up Robust Features, is arguably one of the best detectors for achieving both high efficiency and robustness.

Table 1. Comparison of ORB and SURF Detector on mobile device (Motorola Xoom1) and PC (Thinkpad T420)

Time Detector	Phone (ms)	PC (ms)	Speedup
ORB	170	40	**4x**
SURF	2156	143	**15x**

However, even with enormous advancements in the application processor (AP), the computing power and memory bandwidth of mobile devices are still limited. These limitations are exacerbated when running a SURF detector on a mobile platform, leading to much higher performance overhead than other lighter-weight detectors. As shown in Table 1, running an ORB (Oriented Fast and Rotated BRIEF) detector [3], a state-of-the-art detector designed primarily for efficiency, takes 170ms on a Motorola Xoom1 and 40ms on an i5-based laptop, yielding a 4x speed gap. However, running a SURF detector on them takes 2156ms and 143ms respectively, indicating a 15x speed gap.

Although some light-weight detectors [3, 4, 5] are more efficient than SURF, none of them can match SURF's robustness [22]. They generally cannot achieve satisfactory performance for those mobile applications which demand high accuracy or require handling content with large photometric/geometric changes. There are also several techniques for improving SURF's efficiency by exploiting coherency between consecutive frames [8], employing graphics processing units (GPU) for parallel computing [10] or optimizing various aspects of the implementation [9]. However, none of them analyzes the causes for a SURF detector's poor efficiency on a mobile platform.

We argue that the efficiency capability of a SURF detector on mobile platforms is considerably undervalued. In this paper, we identify and analyze two mismatches between the computations used in the existing SURF algorithm and common mobile hardware platforms, which are the sources of significant performance degradation:

1) *Mismatch between data access pattern and small cache size of a mobile platform.* A SURF detector relies on an integral image and accesses it using a sliding window of successively larger size for different scales. But a 2D array is stored in a row-based fashion in memory (cache and DRAM), not in a window-based fashion; pixels in a single sliding window reside in multiple memory rows (see Figure 1 (a)). The data cache size of a mobile AP, typically 32KB for today's devices, is too small to cache all memory rows for pixels involved in one sliding window, leading to cache misses and cache line replacements and, in turn, incurring expensive memory access.

2) *Mismatch between a huge amount of data-dependent branches in the algorithm and high pipeline hazard penalty of the mobile platform.* To identify a dominant orientation, a SURF detector analyzes gradient distribution around an interest point via a gradient histogram. During this analysis, every pixel around an interest point is mapped to corresponding histogram bins via a set of branch operations. The total number of pixels involved in this

MM'12, October 29–November 2, 2012, Nara, Japan.

Area Chair: Tao Mei

analysis is huge. Thus, the entire process involves an enormous amount of data-dependent branch operations which are not easily predicted. However, the branch predictor and the speculation of out-of-order execution of an ARM (Advanced RISC Machines) based mobile AP are usually not as advanced as that of a laptop or desktop processor. Consequently, it incurs high pipeline hazard penalties, yielding significant performance degradation.

In this paper we propose two techniques, namely *tiled SURF* and *gradient moment based orientation assignment*, to effectively address the above mismatches. Tiled SURF divides an image into tiles (see Figure 1 (b)) and performs point detection for each tile individually to exploit local spatial coherences and reduce memory traffic. Selecting proper tile size is essential to the success of the tiled SURF technique. In this paper we propose a content-aware tile size selection method, based on which a set of heterogeneous tile sizes can be automatically selected to minimize memory traffic and maximize recognition accuracy. To avoid pipeline hazards penalties, we replace the original gradient histogram method with a new branching-free orientation operator based on gradient moments. The two proposed techniques can be seamlessly integrated into a single SURF point detection framework and respectively speed up the two steps (i.e. point detection and orientation estimation) of the process.

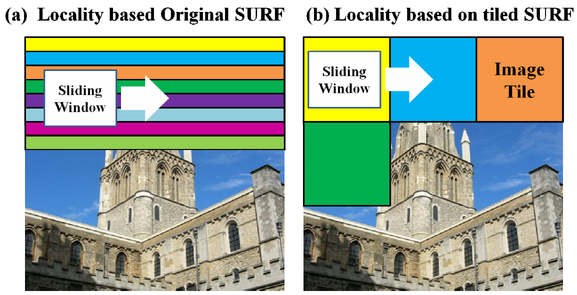
The proposed techniques are evaluated using three mobile platforms. Experimental results demonstrate an overall 6x~8x speedup comparing to the latest OpenCV SURF implementation [14] and 59%~80% reductions in the runtime ratio of the detector running on mobile platforms compared with those on an Intel i5-based *PC*. For mobile recognition tasks with 228 objects, the accelerated SURF achieves similar detection rate and precision compared to the original SURF and 58% improvement relative to an ORB detector.

The following paper is organized as follows: Sec. 2 gives an overview of the related work. Sec. 3 discusses the two causes for the inferior performance of SURF on mobile platforms, and Sec. 4 provides details of two proposed solutions. In Sec.5, we provide empirical results on three mobile platforms for speed and robustness evaluation. Sec. 6 concludes the paper.

2. RELATED WORK

There are several light-weight point detectors designed primarily for high efficiency on mobile devices. Among them, the FAST (Features from Accelerated Segment Test) detector [4, 5] and its variants have been proven quite successful for many real-time mobile applications such as mobile pose tracking [6] and mobile panoramic mapping [7]. FAST is efficient and can find interest points with a reasonable accuracy, but it is neither scale-invariant nor rotation-invariant. To address these limitations, ORB [3] augments FAST with an image pyramid scheme for scale and an orientation operator based on intensity moments. Although FAST and ORB are faster than SURF, both of them are less robust with respect to photometric and geometric changes. In practice, FAST, ORB and SURF provide alternative detector choices to meet specific requirements for various mobile applications.

There have been several successful attempts to speed up SURF. Ta *et al*. [8] proposed a *SURFTrac* algorithm which exploits coherency in video frames to quickly detect interest points at constrained locations. They achieved a 5x speedup compared to the original SURF, but their approach can only be used for applications with continuous image frames. Chen *et al*. [9] proposed a series of implementation optimizations and achieved a 30% speedup over the baseline implementation. In particular, their optimizations include adjusting the sampling rate to reduce computations for local maximum extraction, using approximation for the *arctan* function and reducing floating point computations in the orientation computation phase. Terriberry *et al.* [10] described an efficient SURF implementation on GPUs using OpenCL. However, to date, mobile GPUs only support OpenGL ES and thus porting an implementation from desktop-based GPUs to mobile GPUs remains a tedious task. As mentioned before, none of these techniques identifies the bottlenecks and causes for the slow-down of running a SURF detector on mobile platforms. In this paper, we focus on understanding these reasons and addressing them.



(a) Locality based Original SURF (b) Locality based on tiled SURF

Figure 1. Illustration of data locality and access pattern in (a) the original SURF detector, and (b) the tiled SURF. Each color represents data stored in a unique DRAM row. In the original SURF, a sliding window needs to access multiple DRAM rows, leading to frequent cache misses, while in tiled SURF, all required data within a sliding window can be

Several attempts have been made to improve the performance of mobile computer vision through optimization of mobile hardware. The authors in [12] provided a mobile computer vision benchmarking suite, called MEVBench, to understand mobile vision processing characteristics at the architectural level. Silpa *et al.* [13] attempted to improve the performance by patch memory optimization when using texture memory in mobile GPUs. Clemons *et al.* [11] proposed an embedded heterogeneous multicore design named EFFEX for feature extraction. EFFEX incorporates a new patch memory architecture which stores data in region-order, rather than scan-line order, leading to high spatial locality in data access. Their goal of improving data locality is similar to our tiled SURF. However, they leverage new hardware design, while we rely on software adaptation and optimization.

3. SURF ON MOBILE DEVICES

In this section, we analyze the two causes for the substantial slow-down of running a SURF detector on mobile platforms.

3.1 Mismatch of Data Access Pattern and Small Cache Size of Mobile Platforms

In this subsection, we first review the relevant terms and features of cache miss and reuse. Then we briefly describe the SURF detection process and its data access pattern. After that we discuss why this data access pattern could cause high cache miss on mobile platforms.

3.1.1 Background

A cache miss refers to a failed attempt to read or write a piece of data in the cache, resulting in a main memory access with much longer latency. It can be divided into three categories:

- *Compulsory miss* – Occurs when a cache line is referred for the first time. These misses are inevitable.
- *Capacity miss* – Occurs due to the finite size of the cache. When a program's working set size is larger than the cache size and a cache line containing data that will be reused is replaced before it is reused, a capacity miss occurs when the displaced data is referenced again.
- *Interference miss* – Occurs due to a conflict of cache mapping, i.e. a cache line that contains reusable data is replaced by another cache line that is mapped to the same location. Interference misses on arrays can be divided into two categories: 1) *self-interference misses* that are caused by an element in the same array. 2) *cross-interference misses* that are caused by an element in a different array.

Reuse can be divided into two categories:

- *Spatial reuse* – accesses to the same cache line by references to neighboring data elements.
- *Temporal reuse* – accesses to the same data element multiple times.

3.1.2 Review of SURF Point Detection

A SURF detector selects interest points based on the determinant of Hessian matrix $H(X,\sigma)$:

$$H(X,\sigma)=\begin{bmatrix} L_{xx}(X,\sigma) & L_{xy}(X,\sigma) \\ L_{xy}(X,\sigma) & L_{yy}(X,\sigma) \end{bmatrix}$$

where $X=(x,y)$ is a pixel location in an Image *I*, σ is a scale factor, $L_{xx}(X,\sigma)$ is the convolution of the Gaussian second-order derivative at location *X* and similarly for $L_{xy}(X,\sigma)$ and $L_{yy}(X,\sigma)$.

To speed up the process, a SURF detector approximates the Gaussian second-order derivatives with a combination of box filter responses, computed using the integral image technique. The approximated derivatives are denoted as D_{xx}, D_{xy} and D_{yy} and accordingly the approximate Hessian determinant is:

$$\det(H_{approx})=D_{xx}D_{yy}-(0.9D_{xy})^2$$

A SURF detector computes Hessian determinant values for every image pixel *i* over scales using box filters of successively larger size, yielding a determinant pyramid for the entire image. Then it applies a 3x3x3 local maximum extraction over the determinant pyramid to select interest points' locations and corresponding salient scales. *Pseudo Code 1* shows the process of Hessian determinant pyramid construction. In each iteration of the outer loop S, one layer of Hessian determinant pyramid is constructed based on a reuse of integral image array $I_{integral\_img}$. In the inner loop P, elements of $I_{integral\_img}$ are accessed using a sliding window for computing D_{xx}, D_{xy} and D_{yy} of every image pixel sequentially.

Pseudo Code 1: Determinant Pyramid Construction

```
Input:      scales: total number of scales
            img_pixels: total number of image pixels
Output:     $det_{pyr}$: Hessian determinant pyramid
Variants:   $I_{integral\_img}$: integral image array
            $D_{xx}$, $D_{yy}$, $D_{xy}$: approximate Gaussian second-
            order derivatives in x, y and x-y directions

procedure ConstructDeterminantPyr
  construct integral image $I_{integral\_img}$
  for S = 1: scales
    for P = 1: img_pixels
      access 8 elements of $I_{integral\_img}$ → compute $D_{xx}$
      access 8 elements of $I_{integral\_img}$ → compute $D_{yy}$
      access 12 elements of $I_{integral\_img}$ → compute $D_{xy}$
      compute det → write back to $det_{pry}$[S][P]
    end
  end
```

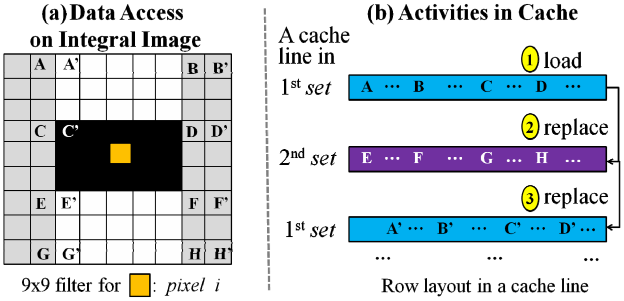


Figure 2. Illustration of (a) one exemplar data access pattern on an integral image, and (b) corresponding activities in cache. In (a) the grey, white and black colors indicate that corresponding values are multiplied by 0, 1 and -2 respectively. Each color in (b) represents a unique set of consecutive rows that can concurrently reside in a cache.

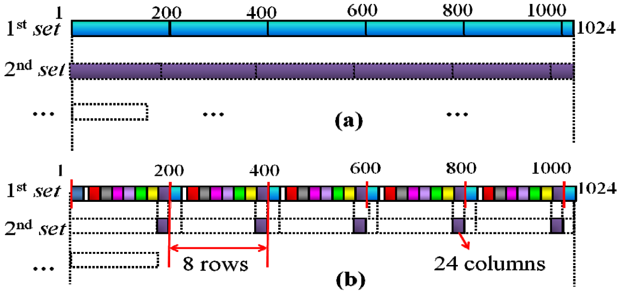


Figure 3. Row layout in a 1024-element cache for (a) 200x200 $I_{integral\_img}$ and (b) tiled $I_{integral\_img,}$ tile size is (8*5+1) rows * 24 columns

3.1.3 Data Access Pattern of SURF Leads to Many Self-Interference Misses

The data access pattern of SURF is determined by the box filter's type and size. For example, a 9x9 box filter for D_{yy} accesses 8 *elements* in a 9x9 rectangle in each iteration, e.g. *element-sets*={*A,B,C,D,E,F,G,H*} in Figure 2 (a). In the next iteration P, the box filter is shifted one pixel to the right and accesses the 8 elements located to the right of *element-set*, e.g. *element-set'* = {*A',B',C',D',E',F',G',H'*} in Figure 2 (a).

We use a simple example to illustrate cache activities, shown in Figure 2 (b), caused by this data access pattern. Consider the layout of a 200x200 array $I_{integral\_img}$ in a direct mapped cache that can hold 1024 elements of $I_{integral\_img}$ (see Figure 3 (a)). Without loss of generality, we assume the first element of $I_{integral\_img}$ falls in the first position of the cache and *element-set* = {*A,B,C,D,E,F,G,H*} is used for the first 9x9 D_{yy} box filter. The 2D $I_{integral\_img}$ array is stored in a row-based order in the cache and *sets* are defined as groups of consecutive rows that can concurrently reside in the cache. In the example of Figure 3 the first *set* consists of rows 1 thought 5 and the first 24 elements of row 6.

As SURF detection starts, the first *set* is loaded into the cache, as shown in Figure 2(b)-①, enabling references to {*A, B, C, D*}, which are located in the 1st and 4th rows. Then to access {*E, F, G, H*} which are in the 7th and 9th rows, the second *set* needs to be fetched from memory and replaces the first *set* (see Figure 2(b)-②). When shifting the box filter one pixel to the right, adjacent elements of {*A, B, C, D*} need to be accessed again (see Figure 2(b)-③). But at this moment, the first *set* has been replaced, and a self-interference miss occurs. In the worst case, every access to the first four elements of a box filter replaces cache lines that contain the latter four elements. Then accessing elements adjacent to the first four elements would cause another self-interference miss. As a result, sequentially computing 9x9 D_{yy} box filter responses based on a 200x200 integral image leads to as many as 80,000 cache misses and cache line replacements. Similarly, computing D_{xx} and D_{xy} would also cause a similar amount of self-interference misses. The self-interference miss would become more significant for both larger box filters and larger images. For instance, computing a 15x15 box filter response on a 200x200 image or a 9x9 filter response on a 400x400 image, each *set* can only contain two elements of a box filter, resulting in twice as many cache misses.

Today's mobile devices, for example Motorola's Xoom1 tablet, typically feature a 32KB L1 data cache, a 512KB L2 data cache and a 1280x800 display. An integral image size of 1281x801 (one pixel larger in width and height to store the sums of all pixels left and above a 1280x800 image) includes 1,026,081 elements. Since each element of an integral image is a 4-Byte integer, 32KB L1 cache and 512KB L2 data cache can only hold up to 8,000 and 128,000 elements, i.e. 16 rows and 99 rows of the array, respectively. As a result, box filters larger than 16x16 and 99x99 would lead to L1 and L2 cache misses in the worst case. In practice, operating systems and other programs inevitably occupy some of the cache, thus there is even smaller available cache size for user applications. As a result even smaller box filters than the above-mentioned would incur cache misses.

3.2 Mismatch of Branches and High Pipeline Hazards Penalty of Mobile Platforms

In this subsection, we first briefly review the orientation computation used in a SURF detector. Then we discuss why this computation incurs a large penalty in mobile platforms.

3.2.1 Review of SURF Orientation Assignment

SURF relies on gradient histograms to identify dominant orientations. First, the entire orientation space is quantized into *N* histogram bins, each of which represents a sliding orientation window covering an angle of $\pi/3$. Then we compute gradient responses of every pixel in a circular neighborhood of an interest point. Based on the gradient orientation of a pixel, we map it to the corresponding histogram bins and add its gradient response to these bins. Finally, the bin with the largest responses is utilized to calculate the dominant orientations of interest points. *Pseudo Code 2* illustrates an OpenCV2.3.1 implementation of this process.

3.2.2 Pipeline Hazard Penalty

A large number of pixels are involved in the orientation analysis. Assuming that an interest point involves ~100 pixels in its neighborhood, then for ~2000 interest points, there would be ~200k pixels involved in total. According to *Pseudo Code 2*, each pixel is mapped to the corresponding histogram bins via a series of branch operations, i.e. "If-then-else" expression. Accordingly, the entire process involves a huge number of branches.

Pseudo Code 2: Orientation by Histogram of Gradient

```
Input:     interest-pts: interest points
           step: orientation increase step
Output:    dominate-oris: dominant orientations
Variants:  $hist_{gradient}$: gradient histogram
           $pixels_{neigh}$: neighboring pixels of interest point
           responses: gradient response array of $pixels_{neigh}$
           oris: gradient orientation array of $pixels_{neigh}$

procedure: ComputeDominateOris
  for Pt = interest-pts
    compute responses
    compute oris
    for Ori = 0: step: 360
      for P = $pixels_{neigh}$
        if (–PI/6 < (oris[P]-Ori) <= PI/6)
          $hist_{gradient}$ [Ori/step] += responses[P]
        endif
      end
    end
    dominate-oris [Pt] = argmax( $hist_{gradient}$ )
  end
```

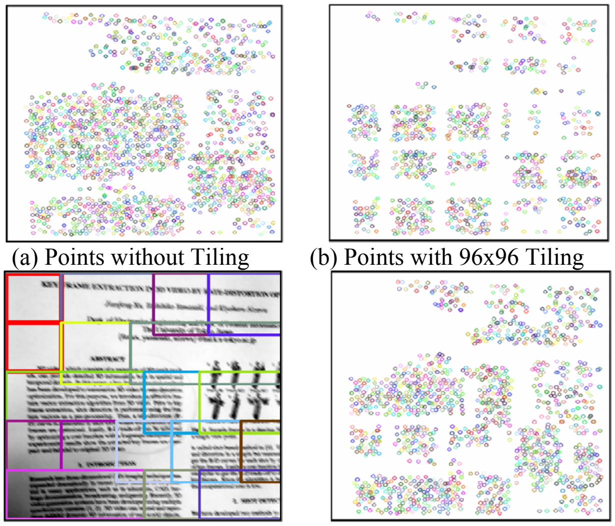
Branches cause control hazards (also called branch hazards) in a hardware pipeline, i.e. when a branch instruction is fetched, the next instruction location is not known until the branch instruction finishes execution. Mis-predicting locations of the subsequent instructions would require flushing the entire pipeline and thus incur a penalty. Modern mobile processors usually use a deep pipeline to increase the CPI, which increase the penalty of each mis-prediction. Moreover, for mobile processors, the branch predictor hardware and other hazard solutions (e.g. out-of-order execution), while improving for each new generation, are still not as sophisticated as desktop and laptop processors (e.g. Intel cores i5 and i7). A conventional SURF implementation involves a huge number of branching operations which cause a substantial penalty due to pipeline hazards and degrade its performance on a mobile platform.

To evaluate the impact of control hazard penalty on the runtime, we implemented another version, i.e. *hist-lookup table*, using a lookup table which stores the correlations between each orientation and the corresponding histogram bins. Without changing the functionality and other computations, the new implementation replaces branches with a lookup table thus avoids control hazards. We compare the time cost of the two implementations on three mobile platforms, Motorola Droid, HTC Thunderbolt, and Motorola Xoom1 tablet.

The results are summarized in Table 2, from which two observations can be made. First, comparing the time cost of *hist-branch* vs. *hist-lookup table* on three mobile platforms and an i5-based laptop, *hist-lookup table* can greatly reduce the runtime by 52% on a Droid, 60% on a Thunderbolt, and 32% on a Xoom1. But it only achieves a 9.6% runtime reduction on a T420 laptop. This result confirms that branch hazard penalty has a much greater runtime impact on a mobile platform than on a PC.

Table 2. Runtime of Orientation Computation based on Gradient Histogram on Mobile Devices and Thinkpad T420

Time Cost (ms)	Droid	Thunder-bolt	Xoom1	ThinkPad T420
hist-branch	2954	1295	717	75
hist-lookup table	1401	518	485	83



(a) Points without Tiling

(b) Points with 96x96 Tiling

(c) Image Overlaid with Content-Aware Tiling

(d) Points with Content-Aware Tiling

Figure 4. Illustration of interested points detected on an image (a) without tiling, (b) with 96x96 tiling, (c) original image overlaid with content-aware tiling, (d) interest points detected on it with content-aware tiling

Therefore, avoiding such penalties is critical for a mobile implementation. Second, we observe that the *hist-lookup table* version achieves larger runtime reduction on the Droid and Thunderbolt than Xoom1. This is because the mobile processors in three devices have different degrees of pipeline hazard penalty. Droid uses a Cortex-A8 processor and Thunderbolt uses a Scorpion, both of which have a 13-cycle branch mis-predict penalty, while Xoom1 features a Cortex-A9 processor that shortens the pipeline to 8 cycles.

While this simple adjustment in implementation to a large degree alleviates the problem, the performance of *hist-lookup table* is still too slow for some mobile applications such as augmented reality. In the following section, we present a new method to further improve SURF's speed. It should also be pointed out that although we use SURF to illustrate the problem, the mismatch also exists in several other computer vision algorithms so the proposed solution is applicable to a broader spectrum of mobile applications.

4. PROPOSED TECHNIQUES

In this section, we introduce two techniques, namely *content-aware tiling based SURF* and *gradient moment based orientation assignment*, to address the mismatches discussed in Section 3.

4.1 Content-Aware Tiling based SURF

Tiling has been extensively studied and applied to graphics rending [17] and complier optimization [18]. In graphics, tiled rendering divides an image into regular grids in the image space to exploit local spatial coherences and reduces the hardware resources required in graphic rendering. In compiler optimization, tiling (also known as blocking) is performed on nested loops to move accesses to reused data closer together in the iteration space to eliminate capacity misses.

Our problem is similar to a loop optimization problem. However, advanced compliers cannot solve our problem because compiler optimization requires loops being tillable, i.e. no cross data references between tiles. For our problem, box filters inevitably access neighboring pixels which may be located in a different tile. Although the loop-permutation and skewing techniques can remove cross-references to some extent, varying cross-reference patterns arising from up-scaling box filters make it impossible for compliers to find a solution for our target problem.

4.1.1 Tiled SURF

Tiled SURF is conceptually similar to tiled rendering and is remarkably simple. We divide an input image into non-overlapping regular grids (i.e. image tiles), and an integral image for each image tile (i.e. integral image tile), is generated. After that, we construct the Hessian determinant pyramid based on each integral image tile and perform point detection and scale selection for every image tile individually.

Tiling reduces the volume of integral image data accessed between reuses of an element, so that the data can remain in the cache for re-access. Figure 3 (b) illustrates the raw layout of a tiled integral image array in a 1024 element cache. In this example, the chosen column size of a tile is 24, so that 8 rows of 24-element columns can fit into a cache space which would otherwise accommodate only one row of a 200-element column. Including the first 24 elements of the 6th row, 41 rows of 24-element columns can fit in the cache without any interference. Therefore, any box filter smaller than 24x41 would not incur any self-interference misses in the 1024-element cache.

4.1.2 Content-Aware Tile Size Selection

The objective of tile size selection is to choose the optimal tile size to minimize memory traffic. The results are platform-dependent and image-size-dependent. We could utilize existing tile size selection algorithms [18] based on a given cache size and cache line size. The problem, however, is that the cache size allocated for user applications is usually smaller than the total cache size and is unknown to application developers. Therefore, using the entire cache size for calculation of the optimal tile size would not be accurate. Empirical selection by running exhaustive experiments on all reasonable tile sizes would be too laborious.

Another problem is that tiling the image would result in loss of interest points at tile borders. This is because a box filter cannot process pixels at borders if at these locations the box filter exceeds the boundary of tiles. Figure 4 (a) and (b) display interest points detected without tiling vs. with 96x96 tiling. Obvious point discontinuities at tile borders can be observed for 96x96 tiling. Overlapping adjacent tiles can compensate interest point loss, but it incurs redundant computational costs due to processing pixels at borders twice.

Intuitively, a larger tile size should incur fewer lost interest points and a smaller accuracy drop than a smaller tile size. But according to a recognition experiment using a set of tile sizes we observe

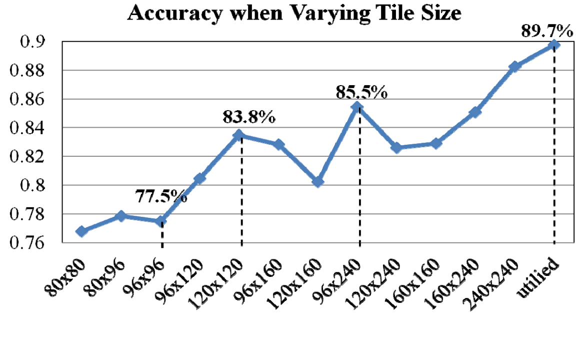


Figure 5. Recognition accuracy when varying tile size

that a larger tile size not always guarantee superior recognition accuracy to a smaller tile size. Figure 5 shows the recognition accuracy of tiled SURF using different tile sizes over 109 images size of 480x480. The tile sizes 120x120 and 96x240 achieve local optima for recognition accuracy. Increasing the tile size from 120x120 to 120x160 or from 96x240 to 160x240, adversely decreases the accuracy. This is because an improper tile size may separate coherent contents (e.g. a continuous text paragraph) into several parts, resulting in informative content residing at tile borders, and the loss of such interest points causes reduction of recognition accuracy. To address this problem, we design a content-aware tile size selection method which uses the following two principles: 1) fitting the reused SURF data into the cache as much as possible, thereby reducing memory access, and 2) minimizing information loss by taking content continuity into consideration.

We use entropy based on intensity probability to represent the amount of information within an image region, where i is an intensity value ranging from 0 to 255 and $P(i)$ is the probability of intensity i within an image region:

$$entropy = -\sum_{0 \le i \le 255} P(i)\log(P(i)) \ . \qquad (1)$$

We define an image region is the *maximum continuous region* (*MCR* for short) if enlarging each sub-region within it can increase the entropy of the sub-region; while further increasing the size of the *MCR* would not further increase the entropy. We select a set of *MCRs* which cover the entire image and each *MCR* in the set is selected as an image tile for the following SURF detection process.

The selection *of MCRs* starts from the top left corner of an image. To reduce the search space, we limit potential tile sizes to a set of pre-determined choices and set the initial size as 96x96. We first compute the entropy for the initial tile. If its entropy is too small, implying little information in the current tile, we skip point detection for this tile. Otherwise, we enlarge the tile to the next larger size and compute the entropy of new tile. The process iterates until the entropy stops increasing as tile size increases or it reaches the largest candidate tile size. The size at which the iteration stops is selected as an *MCR*. Then we move to the next non-overlapping tile and repeat the process. *Pseudo Code 3* shows the selection process of *MCRs*. Figure 4 (c) and (d) displays an exemplar image with content-aware tiling and corresponding detected points on the image. This example exhibits that our method can automatically place larger tiles in long blocks of text, reducing the information loss in these regions. In addition, our method can remove unnecessary processing for non-informative regions (e.g. blank regions), as indicated by the red bold rectangles in Figure 4 (c).

It is worth mentioning that content-aware tiling may also result in a loss of large-scale features, i.e. features whose scale size are larger than the tile size. But according to our experiments on 228 images (details of the dataset please see Section 5.3.2) using the default SURF setting, there are only 0.8% of the features with scale size larger than the smallest tile size 96x96. In addition, among all the large-scale features, only a portion of them can be correctly matched in object recognition tasks. Therefore, the impact of such loss on the final performance is negligible. Object recognition results in Sec. 5.3 will further confirm this argument.

4.2 Orientation by Gradient Moments

In Section 3.2.2, the implementation using a lookup table demonstrates 30%~60% time cost reduction for computing dominant orientations. But updating gradient histograms still takes a significant amount of time. In addition, the lookup table increases memory cost. In this section, we propose an alternative orientation operator based on the gradient moments to further accelerate the SURF detector.

***Pseudo Code 3**: Content-Aware Tile Size Selection*

```
Input:     TSCs : tile size candidates
Output:    MCRs: maximum continuous regions
Variants:  tile-size, $tile\text{-}size_{next}$: tile size before and
                 after increase of this iteration
           entropy, $entropy_{next}$: entropy of a tile
                 before and after increase

procedure SelectMCRs
    Initialize TSCs = {96x96, 96x120,…}
    for each tile
        tile-size = 96x96
        entropy = Entropy( tile-size)
        if (entropy < $\varepsilon$ )
            continue
        for T = 2:end
            $tile\text{-}size_{next}$ = TSCs [T]
            $entropy_{next}$ = Entropy( $tile\text{-}size_{next}$ )
            if ( $entropy_{next}$ > entropy)
                tile-size = $tile\text{-}size_{next}$
                entropy = $entropy_{next}$
            else
                break
            endif
        end
    end
```

The gradient moment [20] of a patch is defined as:

$$mG_{pq} = \sum_{x,y} x^p y^q G(x,y) \ . \qquad (2)$$

where $G(x,y)$ is the gradient responses of location (x,y). With the gradient moments of a patch, we define the *gradient centroid* of the patch as:

$$C_{gradient} = \left(\frac{mG_{10}}{mG_{00}}, \frac{mG_{01}}{mG_{00}} \right) \ . \qquad (3)$$

The *gradient centroid* is analogous to the density centroid of an object, but it is obtained based on the gradient distribution rather than the material density. Much like the density centroid of a non-uniform object is usually offset from its geometric center; the *gradient centroid* of a corner is offset from its center. According to this, we can construct a vector $\overrightarrow{OC}_{gradient}$ from the corner's center O to the *gradient centroid* $C_{gradient}$. $\overrightarrow{OC}_{gradient}$ is determined by the intrinsic characteristic of a corner, thus it is rotation invariant and can be used to compute an orientation. Accordingly, the orientation of the patch is:

$$\theta = \tan^{-1}(m_{01}, m_{10}) = \tan^{-1}\left(\frac{\sum_{x,y} yG(x,y)}{\sum_{x,y} xG(x,y)} \right) \ . \qquad (4)$$

The gradient moment based approach does not involve any branches, nor extra memory overhead arising from a lookup table. Therefore, it completely averts branch hazard penalties and extra memory accesses which are inevitable in the gradient histogram analysis process. In addition, it avoids the time cost of updating the gradient histogram for every pixel. Although the gradient moment based approach incurs two additional multiplication instructions to compute the gradient moments, i.e. multiplying gradients with x and y coordinates, multiplication is reasonably fast for execution, especially with the DSP accelerators commonly available in mobile APs.

Another option for efficiently computing dominant orientations is based on intensity moments, as used in [3]. Computing first order intensity moment only needs one instruction, i.e. multiplying x or y with image intensity $I(x,y)$:

$$m_{pq} = \sum_{x,y} x^p y^q I(x,y) \ . \quad (5)$$

At first glance, this intensity based approach is faster than using gradient moments: computing the gradient responses in x or y direction requires 6 operations based on an integral image. However, the intensity moment based approach utilizes an image pyramid, i.e. a series of down sampled images, for interest points at different scales. While the SURF detector replaces the expensive image pyramid construction phase by processing the original image using up-scaling box filters, no image pyramid is generated in SURF detection. As a result, the intensity moments based approach cannot be employed in the SURF detector.

5. EXPERIMENTS AND RESULTS

In this section, we first describe key features of the mobile platforms used in our experiments in Section 5.1. Then we examine the speedup achieved by each proposed technique in Section 5.2. In Section 5.3, we apply the proposed techniques to mobile recognition tasks. We compare the accelerated SURF with the state-of-the-art ORB detector in terms of recognition accuracy and efficiency.

5.1 Experiment Platforms

We conducted experiments on three mobile platforms: a Motorola Droid, an HTC Thunderbolt, and a Motorola Xoom1 tablet.

The Motorola Droid smartphone runs Android 2.2 Froyo. Launched in Q4 of 2009, Droid' application processor (AP) is TI OMAP 3430 SoC which contains 32KB/32KB L1 instruction/data caches, a 256KB L2 cache and an ARM Cortex-A8 processor running at 600MHz. The Cortex-A8 has a dual-issue in-order 13-stage pipeline and program flow prediction hardware, also known as branch predictor. With program flow prediction enabled, a mis-predicted branch incurs a penalty of 13 cycles.

The HTC Thunderbolt is the first 4G LTE smartphone, which runs Android 2.3.4 Gingerbread. It was launched in Q1 of 2011. Thunderbolt's AP is the second generation 1GHz Scorpion processor from Qualcomm's own design, and has a 13-stage load/store pipeline and two integer pipelines. One of the integer pipelines has 10 stages and can perform simple arithmetic operations while the other has 12 stages and can perform both simple and more complex arithmetic operations. While the cache size of the Scorpion has not been made public, its L1 and L2 caches should be around 32KB and 512KB respectively.

The Motorola Xoom1 is an Android-based tablet, which runs Android 3.0 Honeycomb. It was first introduced at CES in Q1 of 2011. Xoom1's AP is Nvidia's Tegra 2 SoC which features a dual-core ARM Cortex-A9 processor. Each A9 core has its own private 32KB L1 instruction and data caches. The L2 cache is up to 1MB in size, shared by all cores in the AP. The Cortex-A9 has an 8-stage pipeline and follows a dual-issue out-of-order architecture, instead of A8's in-order architecture. While there are multiple cores in the A9, we used only one core in our experiments.

Here, we focus on Android-based platforms. While trying our ideas on iOS-based devices could be interesting, neither approach proposed in this paper utilizes OS-specific functions for the speedup (e.g. setting higher priority for the program in scheduling or calling DMA for fast memory access). Therefore, the trend observed in the results should be OS-independent. In addition, based on the experiments on three different versions of Android OS (see Sec. 5.2) we observe that, the resulting trend is similar and there is no indication that a different OS would make a difference.

To compare the runtime on mobile platforms vs. x86-based PCs, we also run experiments on a Lenovo Thinkpad T420 laptop which is powered by an Intel core i5-2410M running at 2.3GHz.

5.2 Speed Evaluation

5.2.1 Implementation Details

All of our experiments are based on the OpenCV2.3.1 implementation [14] originally released for Windows PCs. Other libraries such as OpenSURF can also be used for the evaluation. But since we compare SURF and ORB, using OpenCV for both detectors provides a more fair comparison: both of them would have similar overhead if any due to other modules of OpenCV. To make a fair comparison between mobile platforms and x86-based PCs, we re-compiled the OpenCV library for all platforms using the same source code.

We tested the runtime of the two proposed techniques using 109 document images of size 480x480. We used the default scale space parameter settings: the initial box filter size is 9x9 and there are 3 octaves, each of which contains 4 layers. Based on these settings, the largest box filter size is 156x156 and on average 2000 points are detected by the original SURF detector.

We used two versions of SURF detector, U-SURF and O-SURF [1]. U-SURF stands for upright SURF, i.e. the dominant orientation is not computed. Since most cache misses occur in the point detection phase, while few happen in the orientation computation phase, we evaluated the tiled SURF based on U-SURF. O-SURF stands for oriented SURF, i.e. dominant orientation is computed. We evaluated the gradient moment orientation operation based on O-SURF.

Computing gradient responses is the basis of SURF point detection and orientation estimation. OpenCV2.3.1 implements this computation in a static function called *icvCalcHaarPattern*, incurring huge call stack overhead. To avoid such overhead and other complier-related issues, we replaced the function call by manually inlining *icvCalcHaarPattern*. In the following part, we denote the original OpenCV2.3.1 implementation as version 1(v1 for short) and the revised version as version 2 (v2 for short).

5.2.2 Evaluation of Tiled SURF

We evaluated tiled SURF using two experiments. In the first experiment, we examine the relationship between the tile size and the runtime on our three mobile platforms. Two metrics are used in this evaluation: 1) *time cost* and 2) *time cost per operation*. We report *time cost per operation* because the number of box filtering operations varies with respect to tile sizes, i.e. a smaller tile size requires fewer operations. Time cost per operation more accurately reflects the overhead resulting from memory traffic.

Six tile-size configurations are considered: 96x96, 120x120, 160x160, 160x240, 240x240 and 480x480. Figure 6 summarizes the results, from which two observations can be made: 1) The

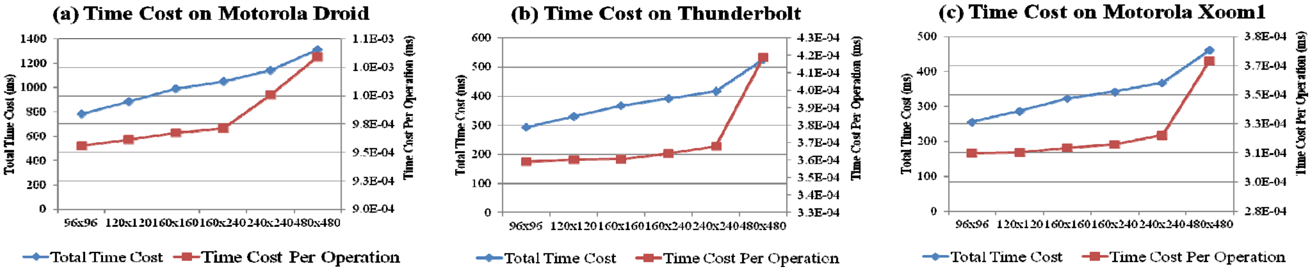


Figure 6. Time cost and time cost per operation of tiled SURF (a) on Droid with Cortex-A8, (b) on Thunderbolt with Scorpion, and (c) on Xoom1 with Cortex-A9

time cost decreases monotonically as the tile size decreases for all three mobile platforms. These results confirm that using smaller tiles can help reduce the time cost. 2) For the curve of time cost per operation, there is a turning point. Increasing the tile size before reaching this point, cost per operation increases slowly; while after this point it increases sharply. The cause of this phenomenon is that when space needed for storing the working set of a tile exceeds the cache size, the number of cache misses increases dramatically, leading to a sudden time cost increase resulting from memory access. We also notice that the turning point varies for different platforms. For a Droid, the turning point is around 160x240, while for a Thunderbolt and a Xoom1, it occurs at 240x240. These results are consistent with the cache sizes of their APs. The Scorpion and the Cortex-A9 have a larger L2 cache than the Cortex-A8, and thus can hold larger data arrays than the latter.

The next experiments focus on examining the speedup achieved by the tiled U-SURF with Content-Aware Tiling (CAT) and how effective it is in narrowing the speed gap between a mobile platform and a x86-based PC. We compare the *time cost* and the *Phone-to-PC ratio* between the original U-SURF and tiled SURF with CAT. The *Phone-to-PC ratio* is the runtime of a program running on a mobile platform divided by that on an x86-based PC, which reflects the speed gap between them.

$$Phone\text{-}to\text{-}PC\ ratio = \frac{\text{runtime on mobile platform}}{\text{runtime on x86-based PC}}$$

The first two rows of Tables 3 and 4 compare the *time cost* and *Phone-to-PC ratio* of U-SURF without and with CAT. The remaining rows of Tables 3 and 4 will be discussed in the next subsection. As expected, U-SURF with CAT can greatly reduce both *time cost* and *Phone-to-PC ratio,* compared to the original U-SURF. At the same time, it reduces the *Phone-to-PC ratio* by 12.5% - 42.9% on the three devices. The reduction in *Phone-to-PC ratio* further indicates that the *mismatch of data access pattern and a small cache size* leads to more severe runtime degradation on mobile platforms than x86-based PCs, so alleviating this problem is critical for performance optimization when porting algorithms to mobile platforms.

5.2.3 Evaluation of Gradient Moment based Method

In this section, we first examine the *Phone-to-PC ratio* and *time cost* for computing orientation alone. Then we integrate the gradient moment based method into SURF and compare the performance of the accelerated SURF with the original SURF, its variant implementations and ORB.

Results of the first experiment are illustrated in Figures 7(a) and (b). First, we compare the *Phone-to-PC ratio* of *hist-branch* and *hist-lookup table* in Figure 7(a) to evaluate branch hazard penalty on runtime for mobile platforms. The results show that *hist-lookup table* greatly reduces the *Phone-to-PC ratio*, demonstrating that the SURF detector running on a mobile platform suffers much greater branch hazard penalty than on an Intel i5-based laptop. Secondly, we investigate the relationship between *Phone-to-PC ratio* reduction and pipeline depth of mobile processors. Since the Cortex-A8 and the Scorpion have a 13-stage pipeline, while the Cortex-A9 has a shorter pipeline of 8 stages, running a SURF detector on the Cortex-A9 has a smaller pipeline hazard penalty and thus avoiding such penalty achieves smaller improvement to *Phone-to-PC ratio*. Thirdly, we inspect the runtime improvement achieved by the *gradient moment*. We compare the absolute time costs of *gradient moment*, *hist-branch* and *hist-lookup table* in Figure 7 (b). The results confirm that *gradient moment* takes the least amount of time among the three. Specifically, it achieves a 14x-28x speedup compared to *hist-branch,* and a 6x-19x speedup compared to *hist-lookup table* on these three mobile devices.

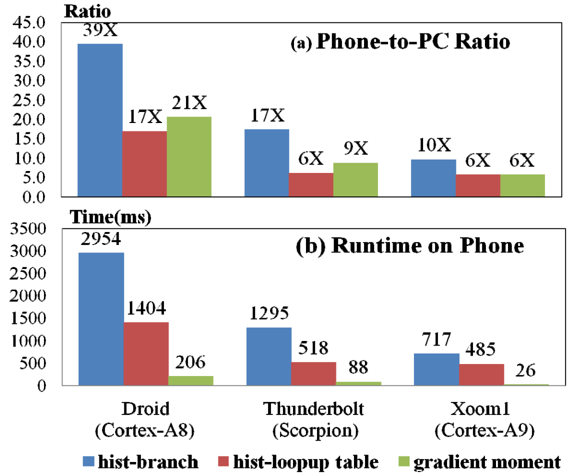


Figure 7. Comparison of (a) Phone-to-PC ratio and (b) runtime on mobile platforms

In the second experiment, we incorporate the gradient moment operator into O-SURF and examine the overall improvement in runtime and *Phone-to-PC ratio* compared to the three versions of the original O-SURF. Runtime and *Phone-to-PC ratio* are shown in the 3rd~6th rows of Tables 3 and 4 respectively. Compared to the original SURF implementation using gradient histograms (i.e. O-SURF v1), the gradient moment based method reduces the overall runtime by 80% on a Droid, 75% on a Thunderbolt and 76% on a Xoom1. At the same time, it improves the *Phone-to-PC ratio* by 65%, 53% and 53% over O-SURF v1, respectively. Finally we apply the two proposed techniques to the O-SURF and compare it with ORB. The results are listed in the last two rows of Tables 3 and 4. Based on the results in 7th and 8th rows of Table 3, with the application of these two proposed techniques, the runtime ratio of running a SURF detector to running an ORB detector on a mobile platform is reduced to 1.6x-1.9x, from 12x-13x if the techniques are not applied. Moreover, the *Phone-to-PC ratio* of the accelerated SURF is reduced to 3x-13x which is similar to that of an ORB detector.

5.3 Mobile Object Recognition

Mobile object recognition is a fundamental task for many embedded computer vision applications, including mobile augmented reality, mobile image search, etc., thus it would be the most effective task to evaluate the performance of the accelerated SURF. In this section, we present results on mobile object recognition to evaluate the robustness of the accelerated SURF.

5.3.1 Implementation Details

We implemented a conventional object recognition pipeline similar to [3]: we first detect interest points and extract descriptors for a query image and then match them to images in a database. We consider the first returned database image with sufficient number of matched points as the result for a query image.

We combine the accelerated SURF detector with two widely used descriptors, the SURF descriptor [1] and the BRIEF descriptor [19], respectively. Each SURF descriptor is a 64-dimensional integer vector; we utilize a k-d tree to fast match the SURF descriptors. The BRIEF descriptor is a 256-bit binary string; correspondingly, we leverage Locality Sensitive Hashing (LSH) for fast matching of the BRIEF descriptors. We use the OpenCV2.3.1 implementation for the descriptors and the indexing structures and set identical parameters for all experiments.

5.3.2 Database and Testing Images

Database [21]: our database consists of two types of images – 109 document images generated from the *ICME06* proceedings and 119 natural scene images containing buildings and people selected from Oxford Building 5K, yielding 228 images in total.

Testing images [21]: we manually captured two pictures for each database image - one of which has ~45^o rotation relative to the database image and is used for testing orientation operators while the other one is upright and used for testing the tiled U-SURF. In mobile applications, images are often blurred due to the motion. In order to examine the robustness of point detectors with respect to such distortions, we implemented motion blur distortions using ImageMagick [15] and applied them to all testing images. Exemplar testing images are shown in Figure 8.

5.3.3 Results

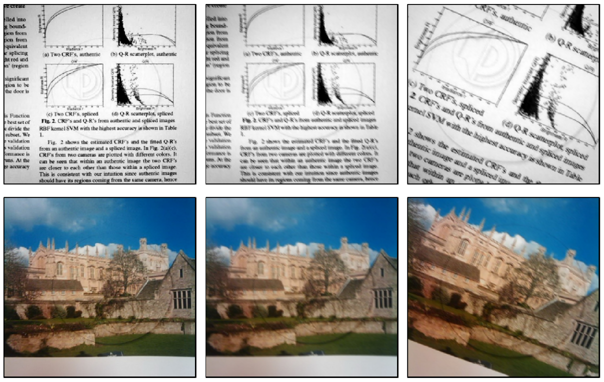
We first focus on investigating the robustness by comparing between: 1) the original SURF vs. the tiled SURF; 2) the original SURF and the tiled SURF vs. the lighter-weight detector ORB. In this experiment we use testing images without rotations. For each detector, we combine it with the SURF descriptor and the BRIEF descriptor, yielding 6 different combinations in total.

Table 3. Time Cost Comparison on Three Mobile Platforms

Time (ms)	Droid	Thunder bolt	Xoom1
U-SURF v2 U-SURF CAT	1310 930	525 356	461 243
O-SURF v1 O-SURF v2 O-SURF Table O-SURF GMoment	7700 4264 2714 1516	2495 1820 1043 613	2156 1178 946 519
O-SURF CAT+GMoment O-ORB [3]	1053 615	404 209	269 170

Table 4. Speed Ratio Comparison on Three Mobile Platforms

Phone-to-PC Ratio (x)	Droid	Thunder bolt	Xoom1
U-SURF v2 U-SURF CAT	20 14	8 7	7 4
O-SURF v1 O-SURF v2 O-SURF Table O-SURF GMoment	54 30 18 19	17 13 7 8	15 8 6 7
O-SURF CAT+GMoment O-ORB [3]	13 15	7 5	3 4



(a) captured Image **(b) motion blur(sigma=8)** **(c) rotation + motion blur**

Figure 8. Exemplar testing images (a) without motion blur, (b) with motion blur, (c) with rotation and motion blur

Table 5. Performance on 228 Testing Images with Rotations

Detector (O-SURF)	Detection Rate	Precision
Untiled + GradHist Untiled + GradMoment CAT + GradMoment	0.963 0.961 0.961	0.963 0.963 0.963

Figure 9 displays the detection rate and precision of the 6 detector-descriptor combinations. Two conclusions can be drawn from the results. 1) Original SURF vs. Tiled SURF: tiling does not degrade on the robustness of a SURF detector, as evidenced by the fact that tiled SURF with CAT achieves similar detection rate and precision as the original SURF. Comparing the curves between "U-SURF untiled+SURF" and "U-SURF CAT+SURF", and between "U-SURF untiled+BRIEF" and "U-SURF CAT+BRIEF", the gap between each of these two pairs of curves is very small. 2) SURF (either original or tiled) vs. ORB: both original SURF and tiled SURF are more robust than ORB with respect to motion blur distortions. As shown in Figure 9, the detection rate and precision of the ORB detector combined with the SURF or BRIEF descriptors decrease sharply with increasing motion blurs, while these two numbers remain almost identical for "U-SURF Untiled+SURF" and "U-SURF CAT+SURF". In comparison, the detection rate and precision of "U-SURF Untilied+BRIEF" and "U-SURF CAT+BRIEF" also drops as motion blur increases, but the slope is still smaller than that of ORB. We think the lack of distinguishing ability for the BRIEF descriptor might account for the drops of these two configurations.

In the second recognition experiment, we compare the performance of the orientation operator using gradient histograms and using gradient moments. In this experiment, we use testing

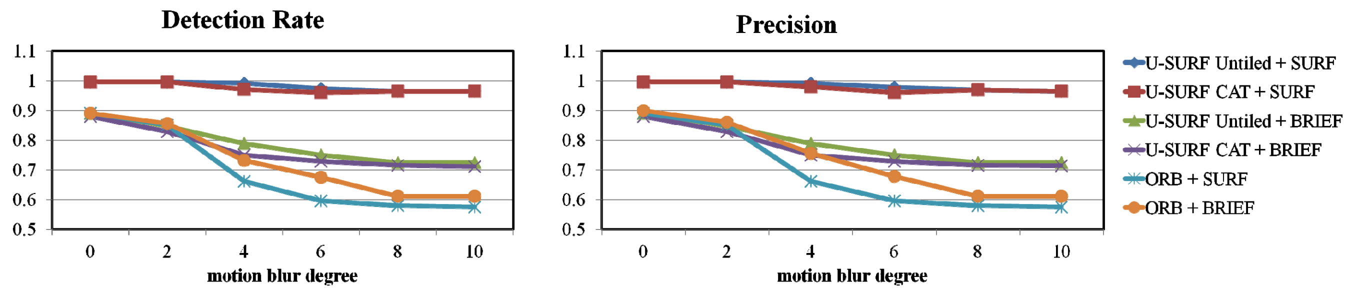


Figure 9. Detection rate and precision on testing images with motion blur distortion of different degrees. We change the blur degree (standard deviation) from 0 to 10 to simulate blur caused by different motion speed.

images with rotations and motion blurs. Table 5 shows the results for testing images whose sigma of motion blur is equal to 8. For other motion blur parameters, the trend is very similar. The first two rows compare the detection rate and precision achieved by the two orientation operators. Both methods achieve very similar performance, demonstrating that the proposed method does not reduce its robustness for producing repeatable dominant orientations. The last row shows the performance achieved by using the two proposed techniques together. The result is consistent with the observations in Figure 9 – the detection rate and precision obtained by using content-aware tiling are very close to those achieved by not using it.

6. CONCLUSION AND FUTURE WORK

In this paper, we analyze the SURF algorithm and identify two mismatches between its required computations and mobile platforms, which result in significant runtime degradation. To address the mismatches, two techniques – content-aware tiling and a gradient moment based orientation operator, are proposed to speed up SURF detection. We successfully demonstrate the high efficiency, high recognition accuracy, and a low *Phone-to-PC runtime ratio* of our accelerated SURF compared to the original SURF detector and the ORB detector.

Despite demonstrating the performance using a single-core mobile AP, the proposed techniques are directly applicable to multi-core APs as well. Future work includes examining the acceleration ability of the proposed methods in parallel computing. It is also worth mentioning that although we focus on SURF detection in this paper, the two identified mismatches may exist in other computer vision algorithms as well. For instance, face detection algorithms which also rely on varying-sized box filers (or Harr features) would suffer from a large number of cache misses due to poor data locality, or decision tree-based algorithms may have high branch mis-prediction penalty arising from many conditional branches while traversing a tree structure. Therefore, adapting existing vision algorithms, which were not designed for embedded systems, to alleviate the problems caused by these mismatches would be the right approach for performance optimization of porting such tasks to mobile platforms.

7. REFERENCES

[1] Bay, H., Ess, A., Tuytelaars, T. and Gool, L.V.. SURF: Speeded-up Robust Features. *In Proc. of ECCV'06.*

[2] Bay, H., Ess, A., Tuytelaars, T. and Gool, L.V.. Speeded-up Robust Features (SURF). *In CVIU,* 110(3), June, 2008.

[3] Rublee, E., Rabaud, V., Konolige, K., and Bradski, G.. ORB: an Efficient Alternative to SIFT or SURF. *In Proc. of ICCV'11.*

[4] Rosten, E. and Drummond, T.. Machine Learning for High Speed Corner Detection. *In Proc. of ECCV'06.*

[5] Rosten, E., Porter, R., and Drummond, T.. Faster and Better: A machine learning approach to corner detection. *In IEEE Trans. PAMI*, 32:105–119, 2010.

[6] Wagner, D., Reitmayr, G., Mulloni, A., Drummond, T. and Schmalstieg, D.. Pose tracking from natural features on mobile phones. *In Proc. of ISMAR'08.*

[7] Wagner, D., Mulloni, A., Langlotz, T., and Schmalstieg, D.. Real-time Panoramic Mapping and tracking on Mobile Phones. *In Proc of VR'10.*

[8] Ta, D.N., Chen, W.C., Gelfand, N., and Pulli, K.. SURFTrac: Efficient Tracking and Continuous Object Recognition using Local Feature Descriptors. *In Proc. of CVPR'09.*

[9] Chen, W.C., Xiong, Y. G., Gao, J., Gelfand, N. and Grzeszczuk, R.. Efficient Extraction of Robust Image Features on Mobile Devices. *In Proc. of ISMAR'07.*

[10] Terriberry, T. B., French, L. M., and Helmsen, J.. GPU Accelerating Speeded-Up Robust Features. *In Proc. of 3DPVT'08.*

[11] Clemons, J., Jones, A., Perricone, R., Savarese, S., and Austin, T.. EFFEX: An Embedded Processor for Computer Vision Based feature Extraction. *In Proc. of DAC'11.*

[12] Clemons, J., Zhu, H.S., Savarese, S., and Austin, T.. MEVBench: A Mobile Computer Vision Benchmarking Suite. *In Proc. of ISWC'11.*

[13] Silpa, B.V.N. and Patney, A. and Krishna, T. and Panda, P.R. and Visweswaran, G.S.. Texture Filter Memory; A Power-Efficient and Scalable Texture Memory Architecture for Mobile Graphics Processors. *In Proc. of ICCAD'08.*

[14] OpenCV2.3.1, http://sourceforge.net/projects/opencvlibrary/

[15] ImageMagick, http://www.imagemagick.org/script/index.php

[16] Lowe, D. G.. Distinctive image features from scale-invariant keypoints. *In IJCV*, 60(2):91–110, 2004.

[17] Foley, J. D., Dam, A. V., Feiner, S. K., and Hughes ,J. F.. *Computer Graphics, Principles and Practice, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1990.

[18] Coleman, S. and McKinley, K. S.. Tile size selection using cache organization and data layout. *In Proc. of SIGPLAN'95.*

[19] Calonder, M., Lepetit, V., Strecha, C., and Fua, P.. BRIEF: Binary Robust Independent Elementary Features. In *Proc. of ECCV'10.*

[20] Rosin, P. L.. Measuring Corner Properties. *In CVIU*, 73(2):291 – 307, 1999.

[21] Yang, X., Liu, Q., Liao, C.Y., and Cheng, K.T.. Large-Scale EMM Identification Based on Geometry-Constrained Visual Word Correspondence Voting. *In Proc. of ICMR' 11.*

[22] Gauglitz, S., Hollerer, T., and Turk, M.. Evaluation of Interest Point Detectors and Feature Descriptors for Visual Tracking. *In IJCV*, 94(3):335-360, 2011.