Accelerating Mobile Augmented Reality on a Handheld Platform

Seung Eun Lee, Yong Zhang, Zhen Fang, Sadagopan Srinivasan, Ravi Iyer, and Donald Newell
Integrated Platforms Architecture, Intel Labs, Hillsboro, OR 97124
{*seung.eun.lee, steven.zhang, zhen.fang, sadagopan.srinivasan, ravishankar.iyer, and donald.newell*}*@intel.com*

***Abstract*— Mobile Augmented Reality (MAR) is an emerging visual computing application for the mobile internet device (MID). In one MAR usage model, the user points the handheld device to an object (like a wine bottle or a building) and the MID automatically recognizes and displays information regarding the object. Achieving this in software on the handheld requires significant compute processing for object recognition and matching. In this paper, we identify hotspot functions of the MAR workload on a low-power x86 platform that motivates acceleration. We present the detailed design of two hardware accelerators, one for object recognition (MAR-HA) and the other for match processing (MAR-MA). We also quantify the performance and area efficiency of the hardware accelerators. Our analysis shows that hardware acceleration has the potential to improve the individual hotspot functions by as much as 20x, and overall response time by 7x. As a result, user response time can be reduced significantly.**

I. Introduction

Smart phones and mobile internet devices (MIDs) have gained widespread popularity by placing compute power and novel applications conveniently in the hands of end users. The introduction of low power general-purpose processors (e.g. Intel®Atom™) expands the capability of MIDs to include disruptive visual computing applications. Emerging visual computing applications such as image/facial recognition, computational photography and motion tracking are quickly entering the mobile domain while nascent disruptive usage cases including virtual worlds and extreme 3D gaming are just around the corner. One disruptive usage model that has become of significant interest to end-users and handheld providers is Mobile Augmented Reality (MAR). Companies such as Nokia, Microsoft and Google are working on and/or have released products to enable MAR in their devices [1], [2], [3].

The MAR workload of interest is best described with an example as follows. Consider a tourist walking the streets of a foreign city and scanning the surroundings using the camera in their smart phone. The smart phone should recognize the objects in the camera image and provide contextual data overlaid on the object in the display. For example, if you are walking in the streets of Jaipur, India and point your MID camera at an interesting building (shown in Figure 1(a)), the MID should display historical information about the building. Similarly, pointing the camera towards a unique object should provide the user with contextual information about it (example shown in Figure 1(b)).

In order to achieve this usage model, the MAR application is required to (a) acquire the image or video stream, (b) recognize objects by computing interest points in the image, (c) match to a pre-established set of images in a database, and (d) display relevant meta-data overlaid on the object in the screen. Among these, steps (b) and (c) are the most computationally challenging tasks, and are the focus of our work. In this paper, we analyze a software implementation of a MAR application and show that the time taken is still in the order of hundreds of milliseconds, which affects user response time and energy consumption. We take the above MAR usage case, analyze the compute requirements, identify the key hotspot functions, and present hardware accelerators for specific key hotspots that help to enable a rich MAR experience on a MID form factor SoC platform. To the best of our knowledge, this is the first work with detailed analysis of the key hotspot functions of MAR and design of hardware accelerator for low power handheld plaforms running MAR.

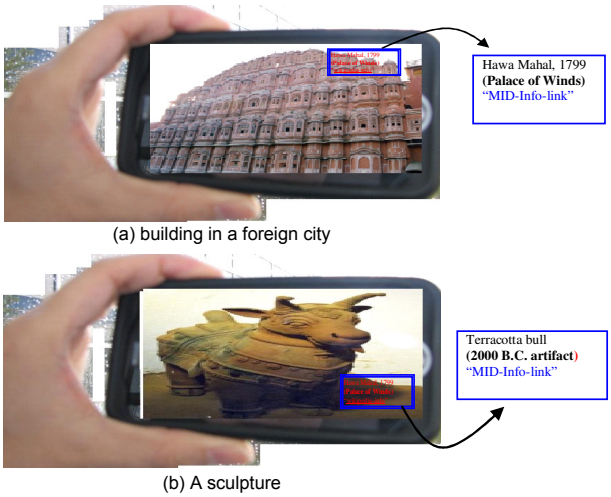

Fig. 1. Mobile augmented reality usage example

The rest of this paper is organized as follows. We first describe the image recognition algorithm that forms the basis of the usage case and provide an overview of the Speeded Up Robust Features (SURF) algorithm [4] and the match algorithm used in the MAR application. Section III describes a set of analysis and software optimizations on MAR. Based on the analysis, we propose two hardware accelerators in Section IV. We discuss the related work in section V, and conclude in section VI by outlining the direction for future work on this topic.

978-1-4244-5028-2/09/$25.00 ©2009 IEEE

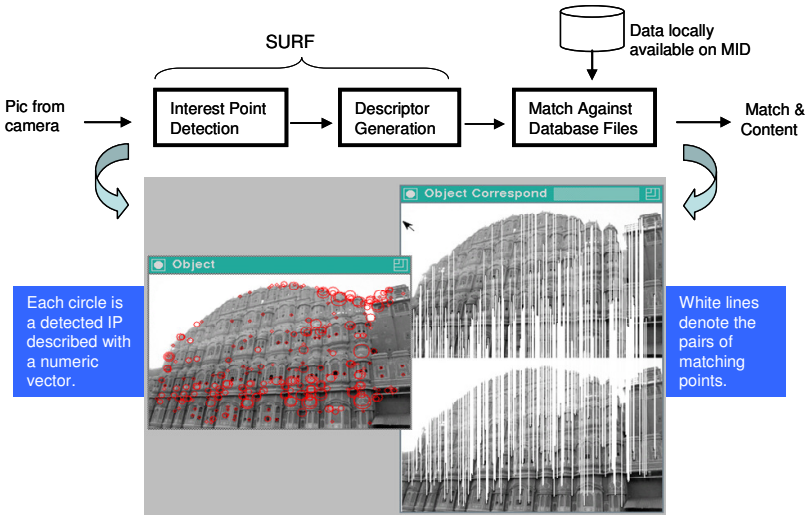

Fig. 2. System flow of mobile augmented reality

II. MAR Application Overview

In the MAR usage scenario, we start with a query image that the user takes with the camera. The intent is to compare this query image against a set of pre-existing images in a database for a potential match. In order to do so, there are three major steps:

- **Interest-point detection**: identify interest points in the query image
- **Descriptor generation**: create descriptor vectors for these interest points
- **Match**: compare descriptor vectors of the query image against descriptor vectors in the database

Figure 2 illustrates the MAR system flow. There are several algorithms that have been proposed to detect interest points and generate descriptors. The most popular algorithms amongst these are variants of SIFT (Scale-Invariant Feature Transform) [5] and SURF (Speeded up Robust Features) [4]. In this paper, we chose the SURF algorithm for our MAR application because it is known to be faster and has sufficient accuracy for the usage model of interest. In addition, researchers have also used SURF successfully for mobile phones for MAR [6]. Below we provide a brief explanation of the SURF algorithm, although we refer the reader to [4] for a more detailed description.

A. Interest Point Detection

SURF uses an interest point detector based on Hessian matrix. Integral image is computed from the input image and speeds up the calculation of any rectangular area. The Hessian matrix is computed for different filter sizes, where the filter size represents the region around which the matrix determinant is computed, with various scale factors. The Hessian computation for a point $X = (x, y)$ in the image at a scale of σ is shown below.

$$H(X,\sigma) = \begin{bmatrix} L_{xx}(X,\sigma) & L_{xy}(X,\sigma) \\ L_{xy}(X,\sigma) & L_{yy}(X,\sigma) \end{bmatrix} \quad (1)$$

where L_{xx}, L_{xy} and L_{yy} are second-order Gaussian derivatives. In SURF, the second-order Gaussian derivates are approximated based on box filters and are denoted as D_{xx}, D_{xy} and D_{yy} instead. The Hessian determinant is computed from these terms as follows:

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

After calculating the Hessian matrix at different scale factors (different octaves, and various filter sizes in each octivees), the interest points are chosen by computing the local maxima (in a $3 \times 3 \times 3$ neighborhood) in scale and image space (i.e. Hessian determinant value of a point is compared against all the neighboring values in the current and neighboring scales).

B. Descriptor Generation

Once the interest points are computed, the next step is to tag the interest point with descriptor vectors. Descriptor vector computes the Haar wavelets, which is the coarse grain pixel contrast of a rectangular region, around a given interest point. In our study, we use 64 element descriptor vector to represent an interest point. The first step in computing descriptor vector is to determine a reproducible orientation. To assign the orientation, Haar wavelet responses are computed in a circular neighborhood around the interest point. The dominant orientation is estimated by calculating the sum of all responses within a sliding orientation window (covering an angle of $\pi/3$). The highest sum of horizontal and vertical responses is chosen as the dominant orientation.

After computing the dominant orientation, a square region of size $20s$ (where s is the scale factor at which the interest point was identified) is chosen around the interest point and oriented along the dominant orientation. The region is split up into smaller 4×4 square sub-regions and Haar wavelets (dx in the x and dy in the y direction) are calculated within each of the sub-regions at regular intervals. The wavelet responses are summed up in each region and the

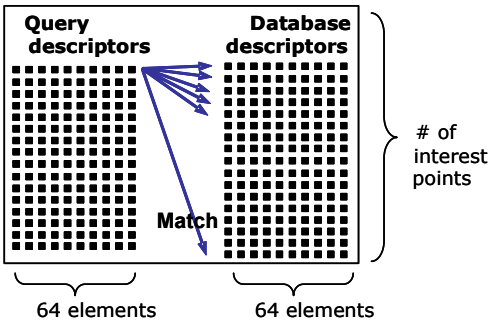

Fig. 3. Brute force match between a query image and a database image

following four dimensional vectors per 4×4 region forms the descriptor vector:

$$V = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|) \quad (3)$$

Such a 64 element descriptor vector is used to describe each interest point in any given image and is used as the basis for the matching step below. We also add the sign of a Laplacian, which is the trace of the Hessian matrix, to the descriptor vector to speed up the matching process.

For interest point detection and description, we used the OpenCV implementation [7] of SURF since it already contains sufficient optimizations as compared to other open source code [8]. The match algorithm is independent of the algorithm used for interest point detection and description. In this paper we employ a brute force match algorithm.

C. Matching

In order to match two images (query and database), we use a brute force match algorithm that exhaustively compares a pair of interest point descriptor vectors from each image based on the Euclidean (a.k.a. L2) distance. Manhattan (a.k.a. L1) distance is another option for the descriptor comparison.

Figure 3 and Algorithm 1 describe how a query image from the camera is matched against a candidate image from the database. One descriptor represents one interest point. The key function is a simple loop shown in the Algorithm 1, assuming 64 elements per descriptor. For each descriptor of the query image, performing the Distance function on all descriptors of a database image gives us the minimum and second minimum ($2^{nd}min$) values of sum. A match for a query descriptor is found in the database image if $min < 0.5 \times 2^{nd}min$. Database images are then ranked based on how many matches they have for the query image, and the highest ranked candidate is selected as the winner.

It should be noted that several other match algorithms are available, such as ANNmatch [9], but the brute force match is simple to implement from a hardware acceleration perspective.

Algorithm 1 Brute Force Matching

```
Match(query,db)
for m = query.ip1 to query.ipM do
  for n = query.ip1 to query.ipN do
    if query.ipm.laplace == db.ip.n.laplace then
      Distance(query.ipm, db.ipn)
    end if
  end for
  Find min and 2nd min for all N Distnace
  if min < 0.5 × 2nd min then
    return query.ipx has a match in the db image
  end if
end for

Distance(Q,D)
for k = 0 to 64 in groups of 4 do
  sum+ = (Q[k] − D[k])^2  //for Euclidean distance
  sum+ = |Q[k] − D[k]|    //for Manhattan distance
  if sum > 2nd min then
    break
  end if
end for
```

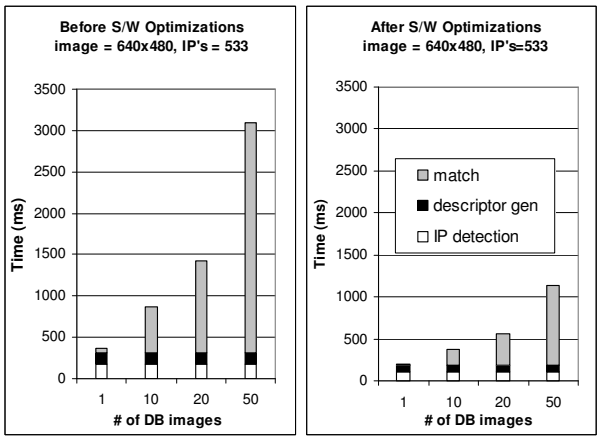

Fig. 4. Software-only performance of MAR on Intel®Atom™

III. Software MAR Performance

We first analyze and optimize the software implementation of MAR on a 1.6GHz Intel®Atom™-based netbook running CentOS4.1 with Linux kernel 2.6. The CPU has hyper-threading enabled supporting 2 hardware threads. Intel®Atom™has 24KB L1 data cache, 32KB instruction cache, and 512KB unified L2 cache. The front-side bus runs at 400 MHz and is connected to 1GB of DDR2-533 DRAM.

Our MAR analysis is based on OpenCV SURF and brute force match implementation. We conducted a detailed analysis of the baseline code, before and after optimizations for Intel®Atom™. While additional algorithm and compiler optimizations may be possible, we use this as our baseline for hardware acceleration analysis. Specifically, for each of the three functions (interest point detection, descriptor generation, and matching), we profiled the application using hardware counter-based performance analysis tools (Vtune). These experiments allowed us to obtain a deep understanding of the performance characteristics, and gain insight to the workload's sensitivity to architectural parameters such as

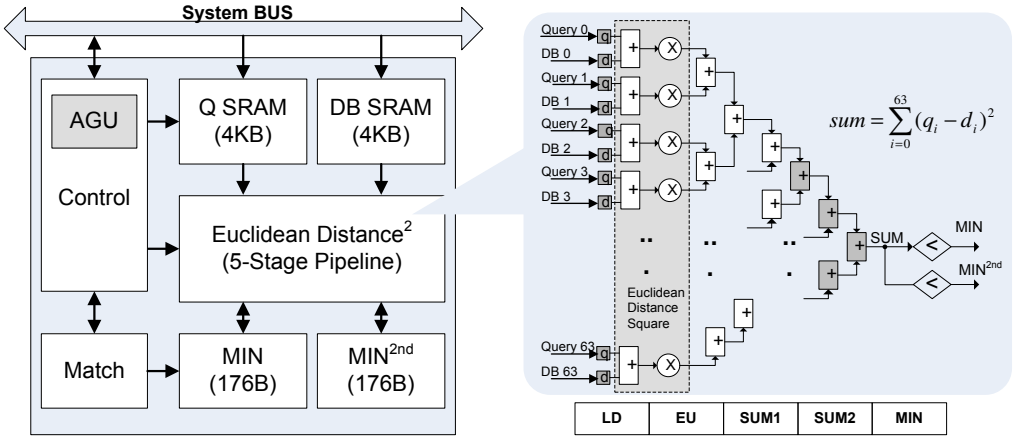

Fig. 5. The microarchitecture of MAR match accelerator

cache size and vector unit width.

Based on our analysis of MAR, we performed a number of optimizations for Intel®Atom™. These optimizations include: cache-aware data padding to reduce conflict misses in the L1D, vectorization, multithreading, basic loop transformation to reduce redundant computations, and precision reduction that allowed us to replace 32-bit floating points with 8-bit integers in the matching function. We refer the reader to [10] for a more detailed analysis. Figure 4 shows an example of the how the optimizations have improved MAR performance on Intel®Atom™. For this set of input, interest point detection time dropped from 183*ms* to 109*ms* (the bottom section of each bar in the figure), descriptor generation 133*ms* to 76*ms* (the middle section of the bars). The time for matching against 20 database images decreased from 1110*ms* to 380*ms*, for example. Matching against 50 database images, the speedup from the applied optimizations is about 3x.

IV. Hardware Accelerators for MAR

We next turn our attention to hardware acceleration analysis to further improve MAR execution time and energy efficiency. The largest fraction of time in software-only processing continues to be the match function (especially when matching against many images in the database). The next most significant hotspot function is interest point detection. For example, when matching against 20 (640×480) images, the match processing takes 67% of execution ($\approx 380ms$) whereas the interest point detection takes 19% of execution time ($\approx 110ms$). In this section, we present two hardware accelerators: a hardware match accelerator (MAR-MA) and a Hessian accelerator (MAR-HA) that computes the Hessian matrix during interest point detection.

A. Match Accelerator (MAR-MA)

The MAR match processing requires computation of the Euclidean distance square or Manhattan distance between every pair of descriptor vectors (one interest point from query image and one interest point from the database image) and computes the number of matches between two images. The MAR match accelerator (MAR-MA) requires the following blocks as shown in Figure 5:

1) SRAM: The MAR-MA accelerator employs SRAM as a staging buffer in the address space, without which all subsequent descriptor values would need to be read from DRAM one at a time. One approach is to support a SRAM size that can accommodate the entire size of $M \times N$ descriptor vectors. For example, if $N = M = 500$, then the size of the SRAM becomes 64KB. However, since an SRAM of this size would consume a significant area, we use a smaller SRAM that supports a 64×64 descriptor computation (requiring only 8KB). The loop in Algorithm 1 is rearranged for 64 descriptor vectors by using loop blocking technique.

2) Euclidean Distance Square: The data-path requires computation of the Euclidean distance square for every pair of descriptor vectors (repalcing multiplier to absolute calculator realizes computation of Manhattan distace). MAR-MA has 5-stage pipeline computing 64 descriptor vectors in 8-bits unsigned integer format concurrently. In LD stage, 64 descriptor vectors are loaded from the internal SRAM into the registers. EU stage calculates the Euclidean distance square between descriptors by computing the difference between a pair of descriptor vector elements and multiplying it with itself to compute the square. SUM1 and SUM2 stages accumulate 64 Euclidean distance squares. Finally, the accumulated value is checked against the minimum and the second minimum value in MIN stage and the results are stored to the internal SRAM for match operation.

3) Control Unit (Loop+AGU): The control unit consists of a set of state machines for indexing calculations as well as address generation for both the computation as well as for fetches issued to DRAM for the next set of operations. First, it fetches a block of query and DB image (64 descriptor vectors for each) to internal SRAM and activates the Euclidean Distance Square data-path. As a result, the minimum and the second minimum for each query vector are stored into MIN and MIN^{2nd} SRAM simultaneously. After the computation for the given blocks, it automatically generates controls over system bus in order to fetch next block of DB image. The Match unit is activated after the completion of computation

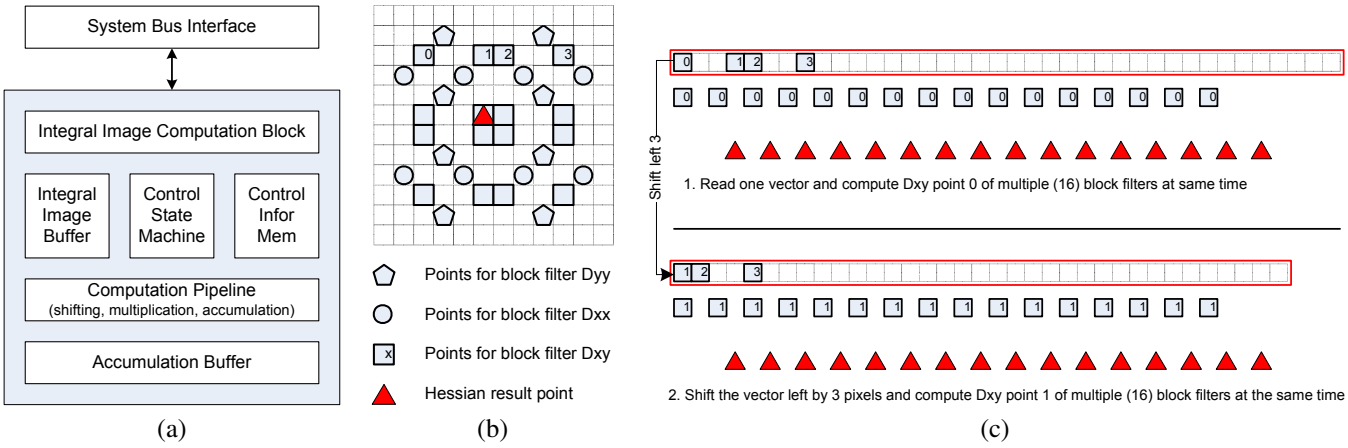

Fig. 6. MAR-HA block diagram and computation illustration

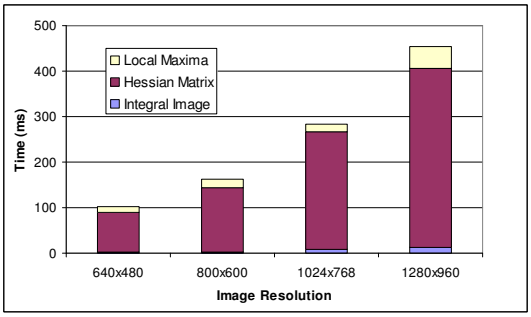

Fig. 7. Execution time breakdown of interest point detection with software optimization

for the given query block and entire DB image to accumulate the number of matches between the block of query and DB image. It continues the operation for entire query image.

4) Match Unit: The final result which is the number of matches between two images is calculated by accessing MIN and MIN^{2nd} memory that are obtained from Euclidean Distance Square unit. When the minimum value is less than half of the second minimum, the number of matches is increased by 1 for the pair of descriptor vectors.

TABLE I
PERFORMANCE AMD MEMORY BW OF MAR-MA AT 400MHZ

SRAM (Q+DB)	Execution Time (ms)	BW (MB/s)
32K	0.275	53.8
16K	0.285	69
8K	0.305	84.5

The performance and memory bandwidth requirements of the match accelerator for two images (308 interest points in each) running at 400MHz are shown in Table I. The performance of MAR-MA accelerator increases with the large SRAM. However, the 8K SRAM demonstrates feasible performance with low area cost. With smaller SRAM, the bandwidth requirement increases due to the multiple accesses for DB image.

B. Hessian Accelerator (MAR-HA)

Interest point detection comprises of three steps which are integral image, hessian matrix, and local maximum computation. Software profiling shows that integral image and Hessian matrix computation take more than 85% of execution time in interest point detection (see Figure 7). Based on this observation, we developed a MAR Hessian accelerator (MAR-HA), as shown in Figure 6(a) to speed up integral image and Hessian matrix computation.

1) Hessian Matrix Computation Description: As described earlier in section II, the calculation of each hessian matrix entry needs the results of 3 box filters, D_{xx}, D_{yy} and D_{xy}. Figure 6(b) shows the 32 integral image points, 8 for D_{xx} and D_{yy} and 16 for D_{xy}, which are required to calculate the box filters for octave 1, octave layer 1. For each integral image point, there is a fixed coefficient, which can be -3, -1, 1 or 3, associated with it. The result of a box filter is the sum of the corresponding integer image points multiply its coefficient. As octave and octave layer goes larger, the shape of these box filters will not change but the size increases and results with an even wider scattered integral image points.

2) Hessian Matrix Computation Exploiting Data Level Parallelism: To optimize the box filter calculation, as shown in Figure 6(a), the block of integral image points covered by these box filters are prefetched into an on-chip buffer. The prefetching and box filter calculation works in a pipelined manner to hide the latency. However even with this on-chip buffer, the box filter integral image point access pattern still makes it quite inefficient and becomes the performance bottleneck. To solve this problem, we proposed a novel approach for hessian matrix computation which provides the following key benefits.

1) Optimal on-chip buffer access pattern.
2) Leveraging data reuse among multiple box filters to further decrease on-chip buffer access.
3) Exploiting data level parallelism among the computation of multiple hessian matrix entries.

The key idea is to compute multiple hessian matrix entries at the same time and store the intermediate results in the Accumulation Buffer. Figure 6(c) shows an example of

calculating 16 hessian matrix entries at the same time and it includes the following steps.

1) Each time, a line of integral image points (we call it vector) are read from on-chip buffer. As shown in figure 6(c), it includes all the second line of integral image points (point 0, 1, 2, and 3 in the figure) for these 16 hessian matrix entries.
2) Shifts this vector left by 1 pixel. Then integral image point 0 for these 16 hessian matrix entries are aligned and ready to be computed by the Computation Pipeline.
3) The shifted vector, coefficient and sum_idx for point 0 are fed to the Computation Pipeline. In Computation Pipeline, with sum_idx as address, an intermediate accumulation vector is read out of Accumulation Buffer and adds with the shifted vector multiply its coefficient. The result is written back into the Accumulation Buffer with shift_value as address.
4) Repeats step 2 and 3 for next point with new shift value, coefficient and sum_idx until the completion of all the points in this line. Then it goes to step 1 and a new vector is read from on-chip buffer. Repeats these steps until all the 32 points are computed. Finally D_{xx}, D_{yy}, D_{xy} are computed based on the value in Accumulation Buffer.

3) Integral Image Buffer Management Scheme: There is a tradeoff between on-chip integral image buffer size and its filling bandwidth (query image is stored in off-chip DRAM before the MAR computation). We choose our buffer management scheme to minimize the on-chip buffer size requirement with only small extra filling bandwidth overhead vs. the ideal case. The width of integral image buffer W_{buf} and height of integral image buffer H_{buf} (with unit of integral image pixel) are

$$W_{buf} = 2(N-1)+S_{filter} \quad (4)$$
$$H_{buf} = S_{filter}+2 \quad (5)$$

where N is the number of hessian matrix entry to be computed at the same time and S_{filter} is the maximum supported block filter size. In our buffer management scheme, the hessian matrix is computed column by column and a column includes N hessian matrix entries. To start with a column, first $W_{buf} \times S_{filter}$ integral image pixels are fetched into integral image buffer and then begins the hessian matrix computation. During the computation, another two lines of integral image pixels are fetched and filled into on-chip buffer. So after the completion of current line of hessian matrix calculation, it can directly go to next line. As the result, the total filling bandwidth B_{fill} is

$$B_{fill} = B_{ideal} \times \frac{2(N-1)+S_{filter}}{2N} \quad (6)$$
$$B_{ideal} = Q_h \times Q_w \times F_{rate} \quad (7)$$

where Q_h is the query image height and Q_w is the query image width and F_{rate} is the supported frame rate. For video of 640×480, 1 byte per pixel, 30 frame per second, N=32 and S_{filter}=30 (6 octave layers), the total on-chip buffer filling bandwidth is 14MB/s and 1.6x of B_{ideal}. While it only requires an on-chip buffer size of 4K integral image pixels and is independent of video resolution.

4) Optimized Integral Image Computation: The function of Integral Image Computation Block is to receive the integral image filling request and respond accordingly. Based on our integral image buffer management scheme, we proposed a novel integral image computation architecture with the following key advantages.

1) Compute integral image on-the-fly and without any start up latency.
2) A fixed latency of less than 15 clock cycles in integral image computation.
3) Minimal buffer size requirement.

The key idea is to compute integral image on-the-fly and keep updating the minimal intermediate states during the computation. For integral image $I(x,y)$, according to the definition, we have

$$I(x,y) = \sum_{i=0}^{i\le x}\sum_{j=0}^{j\le y} img(i,j) \quad (8)$$

We can also rewrite it to

$$I(x,y) = I(x,y-1) + \sum_{i=0}^{i<x-x\%2N} img(i,y) + \sum_{i=x-x\%2N}^{i\le x} img(i,y) \quad (9)$$

$I(x,y-1)$ is stored in the Previous Line Buffer and according to our buffer management scheme, it contains $2(N-1)+S_{filter}$ integral image pixels. $\sum_{i=0}^{i<x-x\%2N} img(i,y)$ is stored in Column Sum Buffer and it contains Q_h integral image pixels. The content of these two buffers are initialized to 0 before computation. Then the integral image computation flow includes the following steps

1) After receiving an integral image filling request of $2(N-1)+S_{filter}$ pixels, Integral Image Computation Block generates request to fetch these pixels from system DRAM.
2) After receiving these pixels from DRAM, $\sum_{i=x-x\%2N}^{i\le x} img(i,y)$ is calculated on-the-fly. With $I(x,y-1)$ and $\sum_{i=0}^{i<x-x\%2N} img(i,y)$ read from Previously Line Buffer and Column Sum Buffer respectively, $I(x,y)$ is calculated, updated into Previous Line Buffer and sent to Integral Image Buffer. After this line is completed, Column Sum Buffer is updated as well.

The total storage is $2(N-1)+S_{filter}+Q_h$ integral image pixels. For video at 640×480, N=32 and S_{filter}=30, the total storage requirement is only 582 integral image pixels.

5) Computing multiple octave layers at the same time: The last optimization we implemented is to compute multiple octave layers at the same time. For the configuration of 6 octave layers, there will be 192 entries in Control Information Memory. This optimization will decrease the off-chip DRAM and on-chip integral image buffer access bandwidth

TABLE II

PHYSICAL CHARACTERASTICS

	MAR-MA	MAR-HA
voltage (V)	0.75	0.75
frequency (MHz)	568	400
Area (mm^2)	0.08	0.36
Dynamic Power (mW)	23.1907	38.3145
Leakage Power (μW)	467.9950	1,823

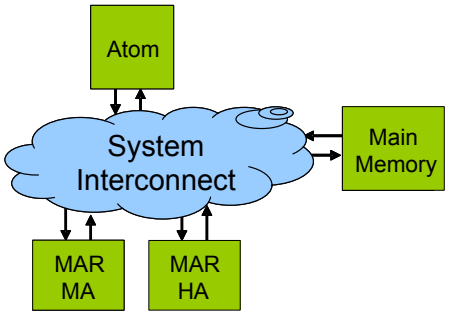

Fig. 8. Block diagram of the MAR platform with hardware accelerators

and improve the hessian matrix computation performance dramatically.

C. Physical Characteristics

The hardware accelerators were implemented using Verilog™HDL and a logic description of our design has been obtained by the synthesis tool from the Synposys™using 45nm technology. Table II summarizes the physical characteristics of the MAR-MA and MAR-HA. The Synopsys™tool chain provided critical path information for logic within the MAR-MA and MAR-HA up to 568MHz and 400MHz, respectively. Two accelerators have an area of approximately 0.08mm^2 and 0.36mm^2 that are feasible for on-chip integration.

D. Performance Benefits

Figure 8 shows the block diagram of MAR platform with hardware accelerators. Software on Intel®Atom™core realizes MAR cooperating with the hardware accelerators (MAR-MA for match and MAR-HA for interest point detection). The core reads the input image and sends the data to MAR-HA which computes integral image and Hessian Matrix. The core identifies the interest points by computing

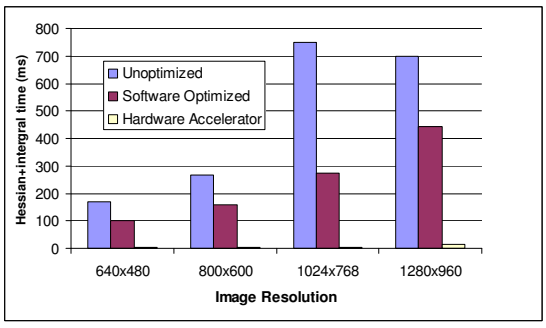

Fig. 9. Execution time of integral image and Hessian matrix computing

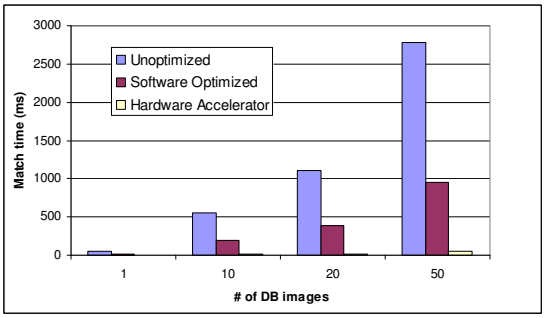

Fig. 10. Execution time of match

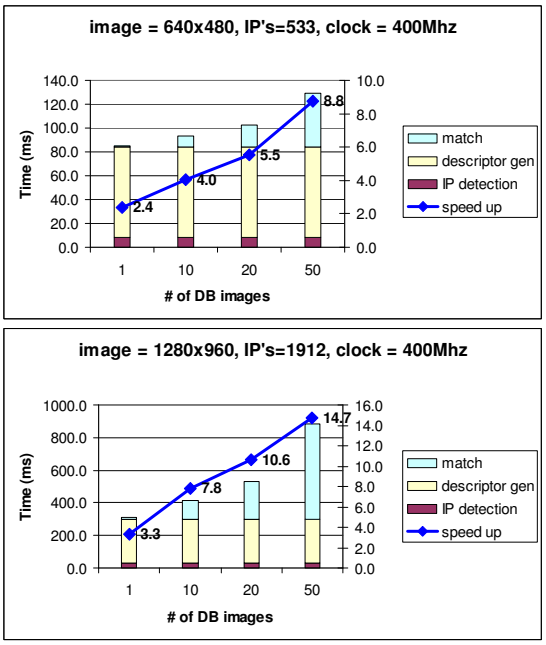

Fig. 11. MAR projected execution time with hardware accelerators

local maxima based on the Hessian matrix calculated by the accelerator. Descriptor generation for the interest points is performed in software as well. Finally, MAR-MA performs match between query and database(DB) images.

The interest point detection is dependent on the number of image pixels. Figure 9 shows the execution time of integral image and Hessian matrix computing as a function of image resolution (i.e. number of pixels per image). We observed that the execution time decreased to 2ms for a 640×480 resolution to 14ms for a 1280×960 image. Overall, interest point detection was optimized by about 13x via the MAR-HA accelerator.

The match portion of the execution time is dependent on the number of database images to match against and correspondingly to the number of interest points to be compared across images. Figure 10 illustrates the execution time of match for 533 interest points. MAR-MA accelerated the processing by 20x compared with software only optimization.

Figure 11 summarizes the performance benefits of the proposed hardware accelerators for MAR processing. As

shown, accelerating MAR by introducing hardware accelerators reduced the overall execution time from 2x to 8x for 640×480 resolution and from 3x to 14x for 1280×960 resolution, reducing the user response time from seconds to milliseconds.

V. Related Work

In recent years, there are few MAR studies primarily focusing on (a) developing algorithms for MAR usage models and (b) analysis of MAR workloads on handheld platforms that are based on ARM or other non-x86 processors. For example, "Touring machine", a prototype MAR device that combines mobile computing with augmented reality was developed in [11]. This is one of the earliest works in MAR that presents information about a location using head-tracked, head-worn, 3D display, and a handheld device.

Feng et al. characterized the SIFT algorithm on multi-core servers [12]. Their work focuses on optimizing the image recognition algorithm for back end servers. Heymann et al. have implemented SIFT on graphics processing unit (GPU) for real-time tracking and recognition of feature points [13]. Sinha et al. have yet another implementation of SIFT on GPU [14]. Both these work try to achieve real-time processing of image recognition algorithm on powerful graphics core, typically seen on workstations.

Recent work has also implemented an outdoor augmented reality system for ARM-based mobile phones in [6]. Zhou et al. focus on matching robust query features against a database [15]. A mobile phone is used to capture the query images and interest points that occur in successive frames are matched against a database that is running on a server. Object based image retrieval using vocabulary of "visual words" to search a large database was done in [16]. Here, the authors use "visual words" as index during query and matching phase. This work focuses on very large dataset as opposed to our small localized database.

Clearly, these previous works highlight that the MAR usage model is gaining significant momentum and such workloads needs to be analyzed for performance and power behavior more seriously. In this paper, we studied one instance of a MAR workload on a Intel®Atom™-based handheld platform and showed that hardware accelerators can improve the performance of the application significantly (by over 7x) over the optimized software implementation.

VI. Conclusions and Future Works

In this paper, we analyzed the execution time of MAR processing on the Intel®Atom™core for SoC Mobile Internet Devices (MIDs) and showed that the base software implementation requires several seconds for MAR processing and therefore requires significant acceleration. We presented several software optimizations to the MAR application (cache, compute, vectorization, multithreading and data type/precision changes), implemented them and showed that these can improve the software processing by as much as 3x.

In order to understand the potential benefits of hardware acceleration, we then proposed hardware accelerators for match processing and interest point detection processing. We presented a detailed design and implementation of the MAR-MA accelerator (for match) and the MAR-HA accelerator (for Hessian computation within interest point detection) and showed that these fairly small (in area) hardware accelerators can improve the overall execution time by as much as 7x over the optimized software implementation.

Future work in this area is as follows. We plan to simulate the hardware accelerators proposed along with the Intel®Atom™core in a SoC platform simulator to study the interaction between the hardware and software components. We also plan to apply the accelerators to MAR processing when dealing with video input (camera panning). We also plan to continue investigating acceleration opportunities for other MAR components (such as descriptor generation). Last but not least, we also plan to study new tightly-coupled accelerator interfaces that allow the user-mode applications running on the core to more effectively communicate with such fine-grain accelerators. We expect that accelerating MAR will bring forth a new spectrum of novel usage models for handhelds.

VII. Acknowledgments

We would like to thank Binu Mathew for some of test images. We would also like to thank Phil Cayton, Jianping Zhou, Madhu Athreya and Jon Tyler for valuable discussions and feedback.

References

[1] Nokia, "Nokia mobile augmented reality," http://research.nokia.com/research/projects/mara/index.html.
[2] "Microsoft techfest 2009," http://www.microsoft.com/presspass/events/msrtechfest/videoGallery.aspx?initialVideo=techfest_channel10_AugmentedReality.
[3] Enkin, http://www.enkin.net.
[4] H. Bay, T. Tuytelaars, and L. Van Gool, "Surf: Speeded up robust features," in *ECCV06*, 2006, pp. 427–434.
[5] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *Int. J. Comput. Vision*, vol. 60, no. 2, pp. 91–110, 2004.
[6] G. Takacs, V. Chandrasekhar, N. Gelfand, Y. Xiong, W.-C. Chen, T. Bismpigiannis, R. Grzeszczuk, K. Pulli, and B. Girod, "Outdoors augmented reality on mobile phone using loxel-based visual feature organization," in *MIR '08*, 2008, pp. 427–434.
[7] OpenCV, http://sourceforge.net/projects/opencvlibrary/.
[8] OpenSURF, http://code.google.com/p/opensurf1.
[9] ANN, http://www.cs.umd.edu/ mount/ANN/.
[10] S. Srinivasan *et al.*, "Performance characterization and optimization of mobile augmented reality on handheld platforms," in *IEEE International Symposium on Workload Characterization*, Oct 2009.
[11] S. Feiner, B. MacIntyre, T. Hollerer, and A. Webster, "A touring machine: prototyping 3d mobile augmented reality systems for exploring the urban environment," in *Wearable Computers, 1997*, pp. 74–81.
[12] H. Feng, E. Li, Y. Chen, and Y. Zhang, "Parallelization and characterization of sift on multi-core systems," in *IISWC 2008*, pp. 14–23.
[13] S. Heymann, K. Mller, A. Smolic, B. Froehlich, and T. Wiegand, "Sift implementation and optimization for general-purpose gpu," in *WSCG*, Jan. 2007.
[14] S. N. Sinha, J.-M. Frahm, M. Pollefeys, and Y. Genc, "Feature tracking and matching in video using programmable graphics hardware," in *Machine Vision and Applications*, March 2007.
[15] Y. Zhou, X. Fan, X. Xie, Y. Gong, and W.-Y. Ma, "Inquiring of the sights from the web via camera mobiles," in *Multimedia and Expo, 2006 IEEE International Conference on*, July 2006, pp. 661–664.
[16] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman, "Object retrieval with large vocabularies and fast spatial matching," in *CVPR '07*, 2007, pp. 1–8.