

# Computer Vision Accelerators for Mobile Systems based on OpenCL GPGPU Co-Processing

Guohui Wang · Yingen Xiong · Jay Yun ·  
Joseph R. Cavallaro

Received: 12 October 2013 / Revised: 27 January 2014 / Accepted: 18 February 2014  
© Springer Science+Business Media New York 2014

**Abstract** In this paper, we present an OpenCL-based heterogeneous implementation of a computer vision algorithm – image inpainting-based object removal algorithm – on mobile devices. To take advantage of the computation power of the mobile processor, the algorithm workflow is partitioned between the CPU and the GPU based on the profiling results on mobile devices, so that the computationally-intensive kernels are accelerated by the mobile GPGPU (general-purpose computing using graphics processing units). By exploring the implementation trade-offs and utilizing the proposed optimization strategies at different levels including algorithm optimization, parallelism optimization, and memory access optimization, we significantly speed up the algorithm with the CPU-GPU heterogeneous implementation, while preserving the quality of the output images. Experimental results show that heterogeneous computing based on GPGPU co-processing can significantly speed up the computer vision algorithms and makes them practical on real-world mobile devices.

**Keywords** Mobile SoC · Computer vision · CPU-GPU partitioning · Co-processing · OpenCL

This paper was partially presented at IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP), Vancouver, Canada, May 2013.

G. Wang (✉) · J. R. Cavallaro  
Department of Electrical and Computer Engineering,  
Rice University, Houston, Texas 77005, USA  
e-mail: wgh@rice.edu

J. R. Cavallaro  
e-mail: cavallar@rice.edu

Y. Xiong · J. Yun  
Qualcomm Technologies Inc., San Diego, California, USA

## 1 Introduction

Mobile devices have evolved rapidly and become ubiquitous over the past decade, giving rise to new application demands through the convergence of mobile computing, wireless communication and digital imaging technologies. On one hand, as mobile processors have become more and more powerful and versatile during the past several years, we are witnessing a rapid growth in the demand for the computer vision applications running on mobile devices, such as image editing, augmented reality, object recognition and so on [3, 6, 11, 20, 21, 25, 30, 31]. On the other hand, with the recent advances in the fields of computer vision and augmented reality, the emerging algorithms have become more complex and computationally-intensive. Therefore, the long processing time due to the high computational complexity prevents these computer vision algorithms from being practically used in mobile applications.

To address this problem, researchers have been exploring general-purpose computing using graphics processing units (GPGPUs) as accelerators to speed up the image processing and computer vision algorithms thanks to the heterogeneous architecture of the modern mobile processors [1, 2, 5, 8, 9, 12, 23, 24, 28]. On desktop computers or supercomputers, numerous programming models have been extensively studied and utilized to facilitate the parallel GPGPU programming, such as the Compute Unified Device Architecture (CUDA) [18] and the Open Computing Language (OpenCL) [13, 15]. As a comparison, due to the lack of parallel programming models in the mobile domain, the OpenGL ES (Embedded System) programming model was commonly used to harness the computing power of the mobile GPU [14]. However, the inherent limitations of the OpenGL ES lead to poor flexibility and scalability, as well as limited parallel performance, due to

the fact that the OpenGL ES was originally designed for 3D graphics rendering. Recently, emerging programming models such as the OpenCL embedded profile [13] and the RenderScript [7] have been supported by the state-of-the-art mobile processors, making the mobile GPGPU feasible for real-world mobile devices for the first time [21, 26, 27].

In this paper, we take the exemplar-based image inpainting algorithm for object removal as a case study to explore the capability of the mobile GPGPU to accelerate computer vision algorithms using OpenCL. The remainder of this paper is organized as follows. Section 2 introduces the architecture of the modern mobile SoCs and the OpenCL programming model for the mobile GPGPU. Section 3 briefly explains the exemplar-based inpainting algorithm for object removal. We analyze the algorithm workflow and propose a method to map the algorithm between mobile CPU and GPU in Section 4. To adapt the complex algorithm to the limited hardware resources on mobile processors, we further study implementation trade-offs and optimization strategies to reduce the processing time in Section 5. Section 6 shows experimental results on a practical mobile device, which indicates that our optimized GPU implementation shows significant speedup, enabling fast interactive object removal applications in a practical mobile device. Section 7 concludes the paper.

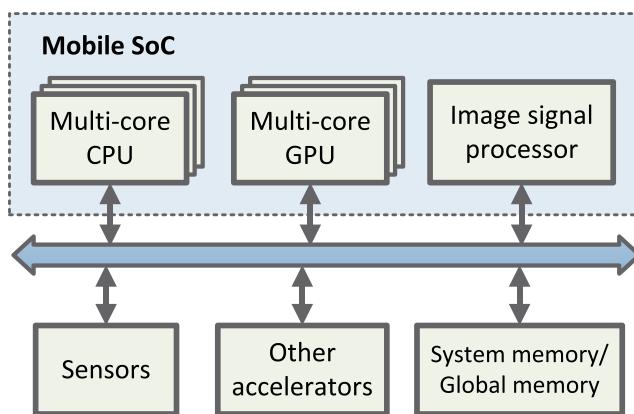
## 2 GPGPU on Mobile Devices

As is shown in Fig. 1, a modern mobile SoC (system-on-chip) chipset typically consists of a multi-core mobile CPU, a mobile GPU with multiple programmable shaders, and an image signal processor (ISP) [10, 19, 22]. Unlike their desktop counterparts, the mobile CPU and GPU share the same system memory via a system bus. The mobile platforms also contain a variety of sensors and accelerators. Modern mobile platforms tend to employ heterogeneous architectures, which integrate several application-specific

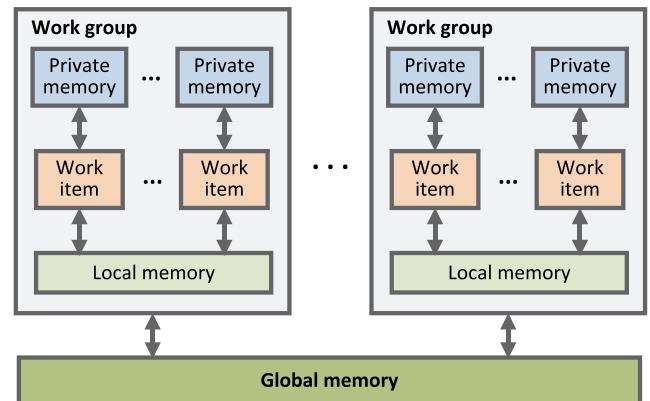
co-processors to enable the computationally intensive algorithms such as face detection and so on. However, the space limitation and the power constraints of the mobile devices limit the number of integrated hardware co-processors. It is preferable to seek the general-purpose computing power inside the mobile processor. The mobile GPUs are suitable candidates to accelerate computationally intensive tasks due to their highly parallel architecture.

The lack of good parallel programming models becomes an obstacle to perform general-purpose computing on the mobile GPUs. As a compromise, researchers have been using the OpenGL ES programming model for GPGPU to achieve performance improvement and energy efficiency on mobile devices during the past decade. For instance, Singhal et al. implemented and optimized an image processing toolkit on handheld GPUs [24]. Nah et al. proposed an OpenGL ES-based implementation of ray tracing, called MobiRT, and studied the CPU-GPU hybrid architecture [16]. Ensor et al. presented GPU-based image analysis on mobile devices, in which the Canny edge detection algorithm was implemented using the OpenGL ES [5]. Researchers have also attempted to accelerate feature detection and extraction using the mobile GPUs [1, 8, 23, 31]. Recently, performance characterization and energy efficiency for mobile CPU-GPU heterogeneous computing have been studied [2, 28].

Thanks to the unified programmable shader architecture and the emerging parallel programming frameworks such as OpenCL, the new generation of mobile GPUs have gained real general-purpose parallel computing capability. OpenCL is a programming framework designed for heterogeneous computing across various platforms [13]. Figure 2 shows the programming and the hierarchical memory architecture of OpenCL. In OpenCL, a host processor (typically a CPU) manages the OpenCL context and is able to offload parallel tasks to several compute devices (for instance, GPUs). The parallel jobs can be divided into work groups, and each of them consists of many work items which are the basic

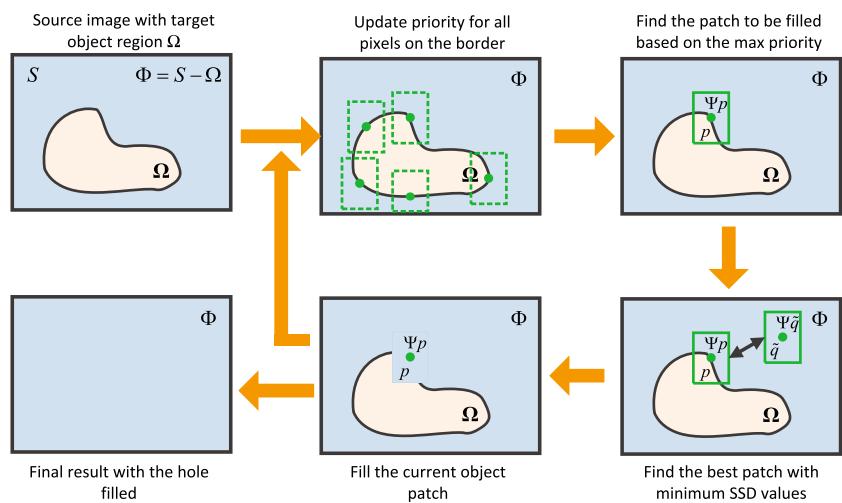


**Figure 1** Architecture of a typical mobile platform.



**Figure 2** OpenCL programming model and hierarchical memory architecture.

**Figure 3** Major steps of the exemplar-based image inpainting algorithm for object removal.



processing units to execute a kernel in parallel. OpenCL defines a hierarchical memory model containing a large off-chip global memory but with long latency of several hundred clock cycles, and a small but fast on-chip local memory which can be shared by work items in the same work group. To efficiently and fully utilize the limited computation resources on a mobile processor to achieve high performance, partitioning the tasks between CPU and GPU, exploring the algorithmic parallelism, and optimizing the memory access need to be carefully considered. Few prior works have studied the methodology of using OpenCL to program mobile GPUs and achieve speedup. Leskela et al. demonstrated a prototype of OpenCL Embedded Profile emulated by OpenGL ES on mobile devices and showed advantages in performance and energy efficiency [12]. In our previous work, we have explored the mobile GPGPU capability of mobile processors to accelerate computer vision algorithms such as an image editing algorithm (object removal), and feature extraction based on SIFT (scale-invariant feature transform) [26, 27]. Performance improvement and energy consumption reduction have been observed.

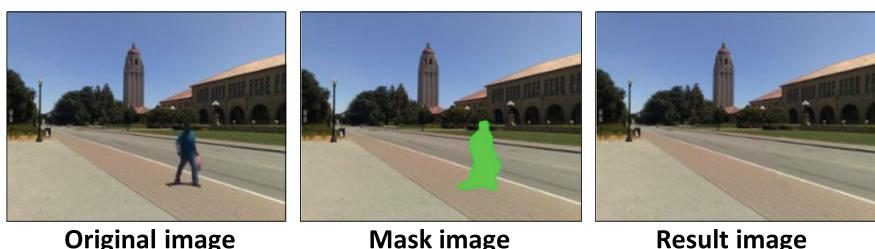
### 3 Overview of Exemplar-based Object Removal Algorithm

In this paper, we take the exemplar-based inpainting algorithm for object removal as a case study to show the

methodology of using the mobile GPU as a co-processor to accelerate computer vision algorithms. The object removal algorithm involves raw image pixel manipulation, iterative image processing technique, sum of squared difference (SSD) computation and so on (Fig. 3), which are typical operations for many computer vision algorithms. Therefore, the case study of the object removal implementation can represent a class of computer vision algorithms, such as image stitching, object recognition, motion estimation, texture analysis and synthesis, and so on. Therefore, by studying and evaluating the performance of the exemplar-based object removal algorithm on mobile devices with CPU-GPU partitioning, the feasibility and advantages of using the mobile GPU as a co-processor can be demonstrated. Furthermore, the optimization techniques proposed in this paper can possibly be applied to other computer vision algorithms with similar operation patterns or algorithm workflows.

Object removal is one of the most important image editing functions. As is shown in Fig. 4, the key idea of object removal is to fill in the hole that is left behind after removing an unwanted object, to generate a visually plausible result image. The exemplar-based inpainting algorithm for object removal can preserve both structural and textural information by replicating patches in a best-first order, which can generate good image quality for object removal applications [4, 29]. In the meanwhile, this algorithm can achieve computational efficiency thanks to the block-based sampling processing, which is especially attractive for a parallel implementation.

**Figure 4** An example of the object removal algorithm. The mask image indicates the object to be removed from the original image.



**Table 1** Specification of the experimental setup.

| Mobile SoC          | Snapdragon 8974     |
|---------------------|---------------------|
| CPU                 | Krait 400 Quad-core |
| Max clock frequency | 2.15 GHz            |
| Compute units       | 4                   |
| Local memory        | 32 KB/compute unit  |
| GPU                 | Adreno 330          |
| Max clock frequency | 400 MHz             |
| Compute units       | 4                   |
| Local memory        | 8 KB/compute unit   |
| Operating system    | Android 4.2.2       |
| Development toolset | Android NDK r9      |
| Instruction set     | ARM-v7a             |

The major steps of the exemplar-based inpainting algorithm for object removal proposed by Criminisi et al. is depicted in Fig. 3 [4]. Assume we have a source image  $S$  with a target region  $\Omega$  to be filled in after an object is removed (the empty region). The left image region is denoted as  $\Phi$  ( $\Phi = S - \Omega$ ). The border of the object region is denoted as  $\delta\Omega$ . The image patches are filled into the object region  $\Omega$  one by one based on priority values  $C(p)$ . Given an image patch  $\Psi_p$  centered at pixel  $p$  for  $p \in \delta\Omega$ , the priority value  $C(p)$  is defined as the product of two terms:

$$P(p) = R(p) \cdot D(p), \quad (1)$$

in which  $R(p)$  is the confidence term indicating the amount of reliable information surrounding the pixel  $p$ , and  $D(p)$  is the data term representing the strength of texture and

**Table 2** Breakdown of execution time for OpenCL kernel functions running only on CPU.

| Kernel functions                                    | Exec time [s] | %             |
|---|---------------|---------------|
| Convert RGB image to gray-scale image               | 0.08          | 0.09 %        |
| Update border of the area to be filled              | 0.60          | 0.69 %        |
| Mark source pixels to be used                       | 0.66          | 0.76 %        |
| Update pixel priorities in the filling area         | 0.45          | 0.52 %        |
| Update pixel confidence in the filling area         | 0.36          | 0.41 %        |
| <b>Find the best matching patch</b>                 | <b>84.4</b>   | <b>97.0 %</b> |
| Update RGB and grayscale image of the filling patch | 0.46          | 0.53 %        |
| Total Time  | 87.0          | 100 %         |

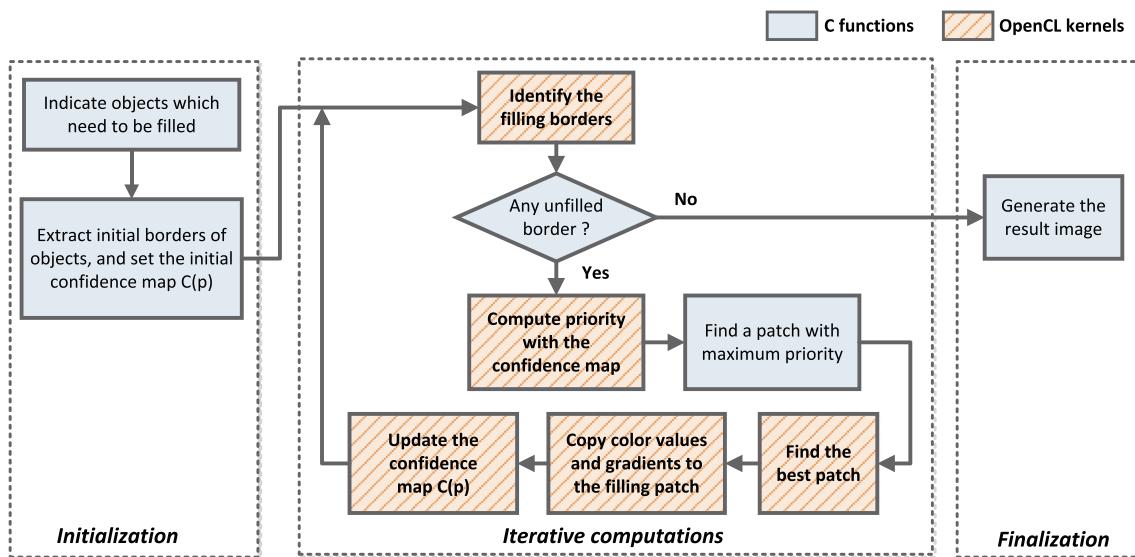
structural information along the edge of the object region  $\delta\Omega$  in each iteration.  $R(p)$  and  $D(p)$  are defined as follows:

$$C(p) = \frac{\sum_{q \in \Psi_p \cap \Omega} C(q)}{|\Psi_p|},$$

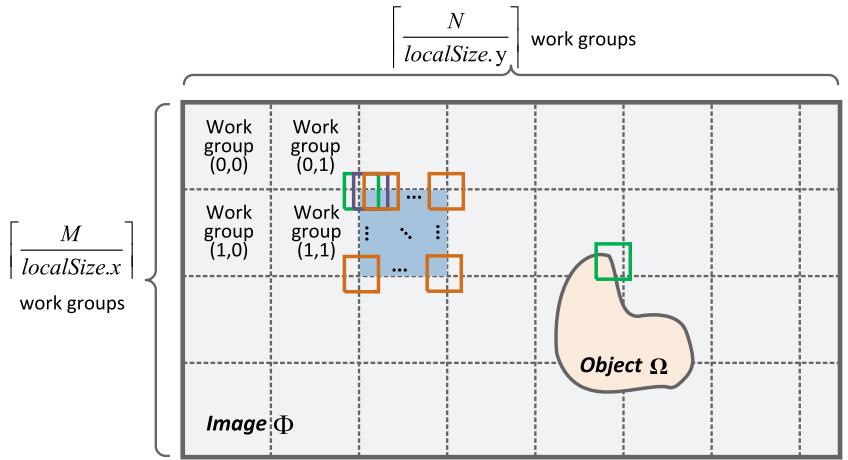
$$D(p) = \frac{\nabla I_p^\perp \cdot n_p}{\alpha}, \quad (2)$$

where  $|\Psi_p|$  is the area of  $\Psi_p$ ,  $\alpha$  is a normalization factor (for a typical grey-level image,  $\alpha = 255$ ), and  $n_p$  is a unit vector orthogonal to  $\delta\Omega$  in the point  $p$ .

According to the priority values for all patches across the border  $\delta\Omega$  of the target region, we select a candidate patch with the maximum priority value. Then, we search the image region  $\Phi$  and find a patch  $\Psi_{\tilde{q}}$  that best matches the patch  $\Psi_p$  (we call this step *findBestPatch*). The goal of *findBestPatch* is to find the best matching patch  $\Psi_{\tilde{q}}$  from candidate patches  $\Psi_q$  in the source image region

**Figure 5** Algorithm workflow of the exemplar-based object removal algorithm. Please note that one OpenCL block might be mapped to one or multiple OpenCL kernels depending on the computation and memory data access patterns.

**Figure 6** Algorithm mapping of *findBestPatch* kernel function using OpenCL.



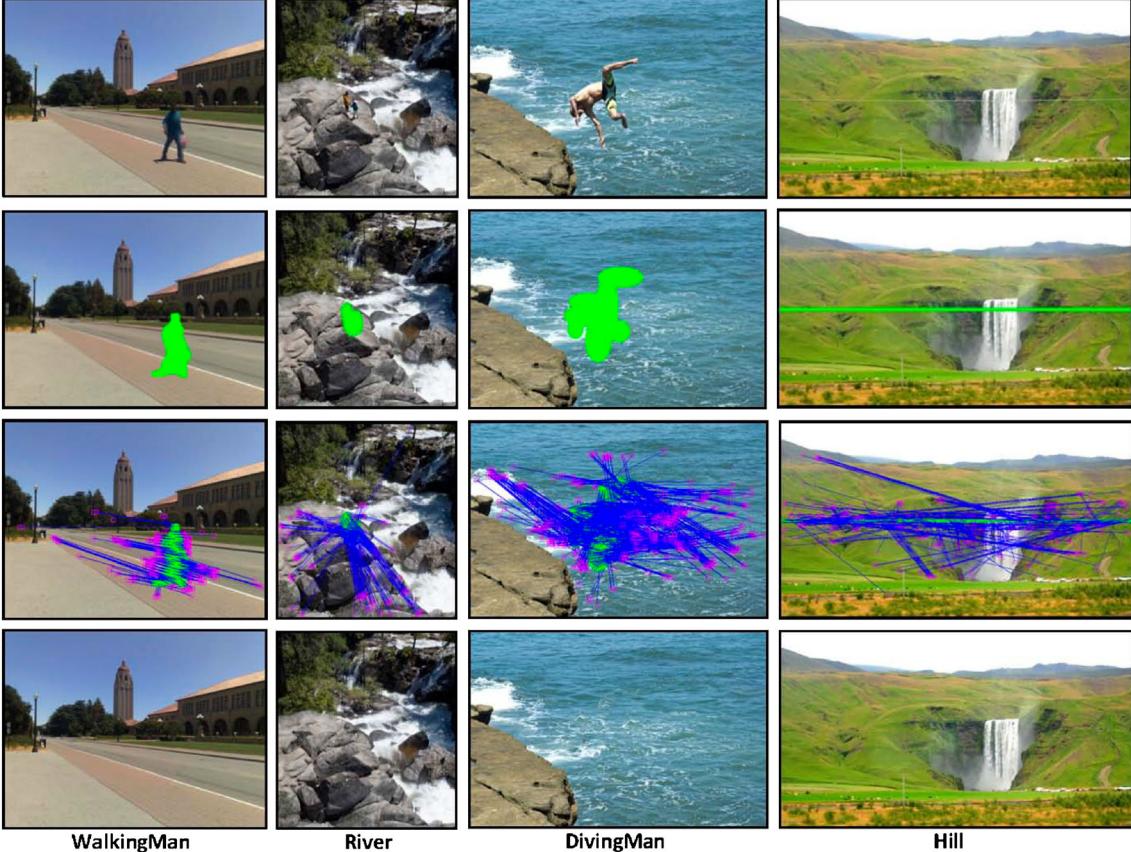
$\Phi$ , to match an object patch  $\Psi_p$  in the object region  $\Omega$  based on a certain distance metric. The sum of squared differences (SSD) is typically used as a distance metric to measure the similarity between the patches [4]. We denote the color value of a pixel  $x$  by  $I_x = (R_x, G_x, B_x)$ . For an object patch  $\Psi_p$ , the best patch  $\Psi_{\tilde{q}}$  is chosen by computing:

$$\Psi_{\tilde{q}} = \arg \min_{q \in \Phi} d(\Psi_p, \Psi_q), \quad (3)$$

in which  $d(\Psi_q, \Psi_p)$  is defined as follows:

$$d(\Psi_q, \Psi_p) = \sum_{p \in \Psi_p \cap \Phi, q \in \Psi_q \cap \Phi} (I_p - I_q)^2. \quad (4)$$

Assume that the size of the original image is  $M \times N$ , and the size of the patch is  $P \times P$ . The complexity of *findBestPatch* can be estimated as  $O(MNP^2)$ . Based on our profiling (will



**Figure 7** The best patch mapping found by a full search and the result images. The 1st row: original images. The 2nd row: masks covering the unwanted objects. The 3rd row: best patch mapping; the

small squares indicate the best patches found by the *findBestPatch()* function. The 4th row: result images.

**Table 3** Description of images in the test dataset. These images are selected to represent different types of image scenes and different shapes of objects to be removed.

| Image      | Image type             | Image size | Object type      | Object size |
|------------|------------------------|------------|------------------|-------------|
| WalkingMan | Complex scene          | 512 × 384  | Blob             | 78 × 126    |
| River      | Complex scene, texture | 480 × 639  | Small blob       | 59 × 93     |
| DivingMan  | Texture                | 576 × 384  | Big/random shape | 155 × 186   |
| Hill       | Complex scene          | 1024 × 550 | Long strip       | 1024 × 10   |

be shown in Section 4), *findBestPatch* compute kernel occupies the most computations in the whole object removal algorithm.

Once the best matching patch is found, we copy the pixel values of  $\Psi_{\tilde{q}}$  into  $\Psi_p$ . The aforementioned search and copy process is repeated until the whole target region  $\Omega$  is filled up. More details of the algorithm can be found in reference [4].

#### 4 Algorithm Workflow Analysis and CPU-GPU Partitioning

In this section, we analyze the workflow of the object removal algorithms and describe the algorithm partitioning between the CPU and GPU to fully utilize the resources of the mobile SoC chipset.

##### 4.1 Experimental Setup

The profiling and experiments are performed on a development platform consisting of a Qualcomm Snapdragon 8974 chipset [22], which supports OpenCL Embedded Profile for both CPU and GPU. The details of the experimental setup are listed in Table 1.

##### 4.2 Algorithm Mapping

Figure 5 shows a workflow diagram of the exemplar-based inpainting algorithm for object removal. The algorithm can

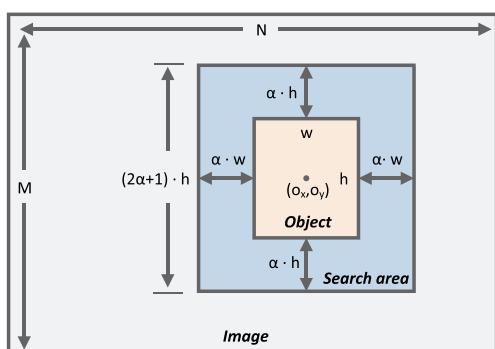
be partitioned into three stages: initialization stage, iterative computation stage, and the finalization stage. The blocks with the slashed lines are core functions inside the iterative stage and represent most of the computational workload. We can map the core functions into OpenCL kernels to exploit the 2-dimensional pixel-level and block-level parallelisms in the algorithms. The CPU handles the OpenCL context initialization, memory objects management, and kernel launching. By analyzing the algorithm, we partition the core functions into eight OpenCL kernels based on the properties of the computations, as is shown in Table 2. In each OpenCL kernel, the fact that no dependency exists among image blocks allows us to naturally partition the tasks into work groups. To represent color pixel values in RGBA (red green blue alpha) format, we use efficient vector data structures such as *cl uchar4* to take advantage of built-in vector functions of OpenCL.

To better optimize the OpenCL-based implementation, we first measure the timing performance of the OpenCL kernels. Table 2 shows a breakdown of processing time when running the program on a single core of the CPU on our test device. The OpenCL kernel function used to find the best matching patch with the current patch (denoted as *findBestPatch*) occupies most of the processing time (97 %), so optimizing the *findBestPatch* kernel becomes the key to improving performance.

#### 5 Algorithm Optimizations and Implementation Trade-offs

##### 5.1 OpenCL Implementation of *findBestPatch* Kernel Function

The algorithm mapping of *findBestPatch* kernel used for the OpenCL implementation is shown in Fig. 6. To perform a full search for the best patch  $\Psi_{\tilde{q}}$  to match the current filling patch  $\Psi_p$  in the *findBestPatch* OpenCL kernel, we spawn  $M \times N$  work items, with each computing an SSD value between two  $P \times P$  patches. We partition these  $M \times N$  work items into work groups according to the compute capability



**Figure 8** Diagram of reduced search area with search factor  $\alpha$ .

of the GPU. The size of 2-dimensional work groups can be expressed as

$$(\lceil M/localSize.x \rceil, \lceil N/localSize.y \rceil). \quad (5)$$

In our implementation, each work group contains  $8 \times 8$  work items ( $localSize.x = 8, localSize.y = 8$ ). Therefore, each work group of work items perform SSD computations for 64 patch candidates. The parallel implementation of the SSD computation in the *findBestPatch* kernel function is detailed in Algorithm 1.

---

**Algorithm 1** Compute SSD values between all candidate image patches and the image patch to be filled using an OpenCL kernel function.

---

```

1. Input:
  (a) Position of object patch  $\Psi_p$ :  $(p_x, p_y)$ ;
  (b) Patch size  $P$ ;
2. Output: SSD array  $ssdArray$ ;
3. Begin OpenCL kernel:
4.   Get global ID for the current work item:  $(i, j)$ ;
5.   float  $sum = 0.0$ ;
6.   int  $src_x, src_y$ ; // source pixel position
7.   int  $tgt_x, tgt_y$ ; // target pixel position
8.   int  $winsize = P/2$ ;
9.   for(int  $h = -winsize; h \leq winsize; h ++)$ 
10.    for(int  $w = -winsize; w \leq winsize; w ++)$ 
11.       $src_x = i + w; src_y = j + h;$ 
12.       $tgt_x = p_x + w; tgt_y = p_y + h;$ 
13.      if(( $src_x, src_y$ ) or ( $tgt_x, tgt_y$ ) is out of image)
          continue;
14.      end if
15.      if(pixel ( $tgt_x, tgt_y$ ) is inside source region  $\Phi$ )
          Read pixel ( $tgt_x, tgt_y$ ) data into  $tgtData$ ;
18.          Read pixel ( $src_x, src_y$ ) data into  $srcData$ ;
19.           $sum += (tgtData - srcData)^2$ ;
20.      end if
21.    end for
22.  end for
23.  Store  $sum$  into  $ssdArray$ ;
24. End OpenCL kernel
```

---

## 5.2 Experimental Dataset

To demonstrate the best patch matching behavior of the object removal algorithm, we employ several different test images to cover different scenarios, which are shown in the first row of Fig. 7. The characteristics of these test images are summarized in Table 3. We chose images with different background scenes and textures, with variable image sizes and object sizes, and with different object shapes, so that

by performing experiments on these images, we can better understand the implementation trade-offs related to the performance improvement.

## 5.3 Reducing Search Space

We have done experiments to verify the locations of the best patches found by a full search across the whole image area. For the test images shown in Fig. 7, most of the best patches are found in a region surrounding the object area. The reason is that adjacent areas usually have similar structures and textures in natural images. To reduce the searching time, we can utilize this spatial locality by limiting the search space. To better model this optimizing strategy in a scalable manner, we define a search factor  $\alpha$ . Assume the width and height of the object area are  $w$  and  $h$  respectively. The new search area is formed by expanding the object area by  $\alpha h$  to the up and down directions, and  $\alpha w$  to the left and right directions, as is shown in Fig. 8. The search area factor  $\alpha$  has a range of  $0 \leq \alpha < \max(M/h, N/w)$ . Assume the object region is centered at coordinate  $(o_x, o_y)$ . Figure 8 shows a typical case in which the whole search area is inside the image area. For more general cases, the boundary of the new search area becomes:

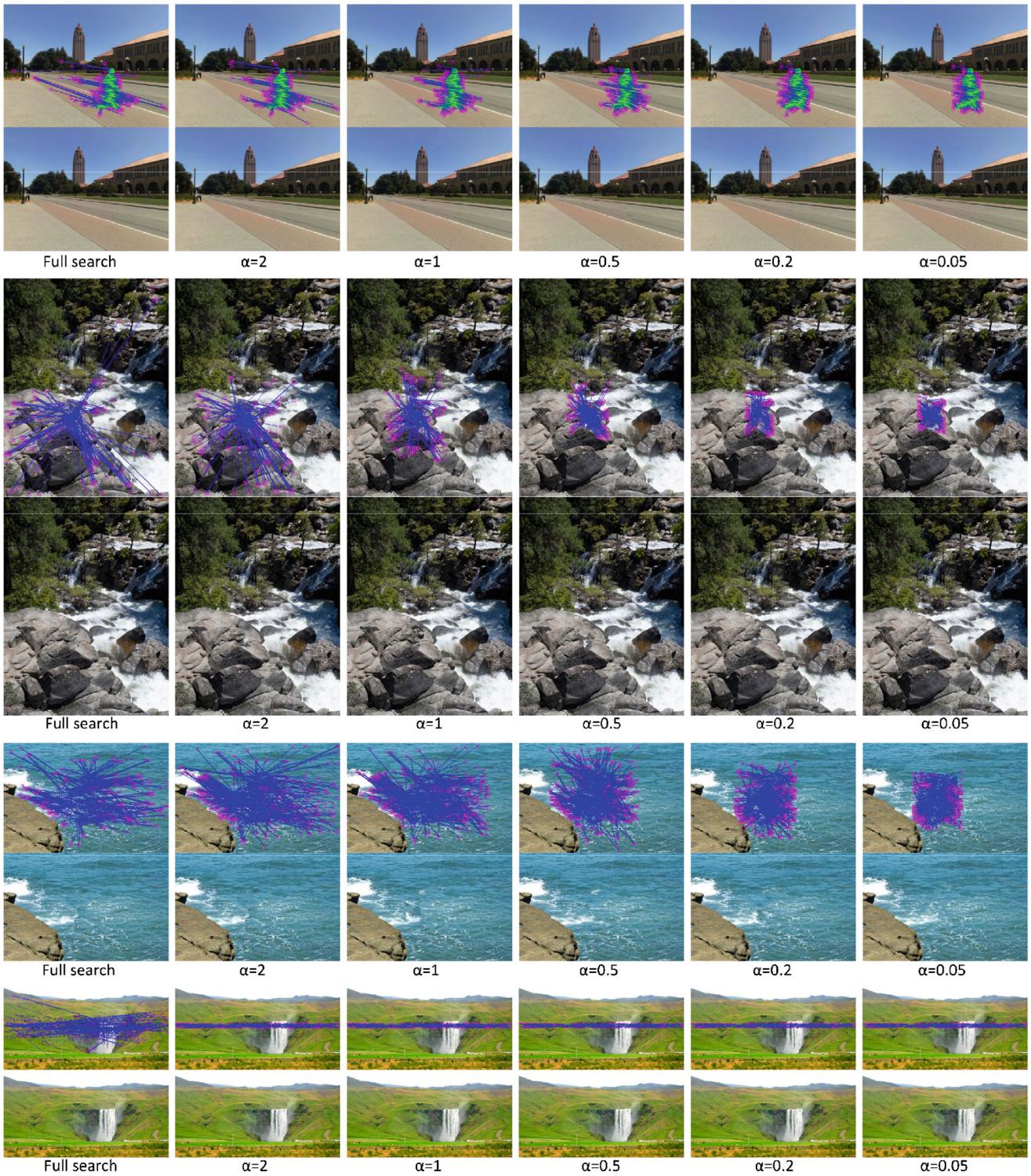
$$\begin{aligned} B_{left} &= \max\left(0, o_x - \left(\frac{1}{2} + \alpha\right)w\right), \\ B_{right} &= \min\left(N, o_x + \left(\frac{1}{2} + \alpha\right)w\right), \\ B_{top} &= \max\left(0, o_y - \left(\frac{1}{2} + \alpha\right)h\right), \\ B_{bottom} &= \min\left(M, o_y + \left(\frac{1}{2} + \alpha\right)h\right). \end{aligned} \quad (6)$$

By defining the search factor  $\alpha$  this way, we can easily adjust the search area. Moreover, this method allows the search area to grow along four directions with an equal chance, so as to increase the possibility of finding a better patch. Since there are no useful pixels in the object area for patch matching, only the candidate patches not in the object region will be compared with the object patch. So the actual size of the search area ( $SA$ ) can be expressed as:

$$\begin{aligned} SA &= (2\alpha + 1)w \cdot (2\alpha + 1)h - wh \\ &= ((2\alpha + 1)^2 - 1)wh. \end{aligned} \quad (7)$$

The complexity of *findBestPatch* can be estimated by  $O(((2\alpha + 1)^2 - 1)whP^2)$ . Thus, when we reduce the search area (reducing  $\alpha$ ), the complexity to search the best patch reduces significantly.

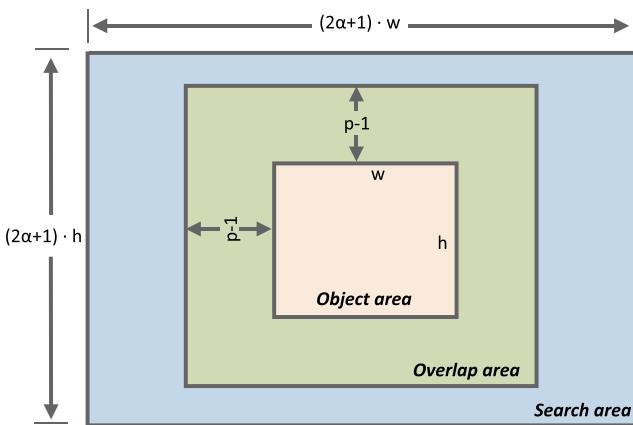
Figure 9 demonstrates the effect of reducing the search factor  $\alpha$ . After reducing the search factor, the best matching patches are limited to a small region around the object region. Even when the search factor is reduced



**Figure 9** The effect of reducing the search area. The search area factor  $\alpha$  is defined as in Fig. 8.

significantly to only  $\alpha = 0.05$ , we still get visually plausible results. Due to its importance, choosing a proper search factor  $\alpha$  is critical for practical applications to achieve good performance and efficiency. Based on our experiments, for a regular object region, choosing

parameter  $\alpha$  in the range of  $0.05 \leq \alpha < 0.5$  normally leads to good implementation trade-offs. If the algorithm is to be applied to a certain type of images, trainings on datasets can be performed to determine a proper search factor  $\alpha$ .



**Figure 10** The relationship among search area, object area and the overlap area for the best patch searching.

In addition to time reduction, reducing the search area can also reduce the possible false matching. As a comparison metric, SSD can roughly represent the similarity of two patches, but it cannot accurately reflect the structural and color information embedded in the patches. Therefore, the patches with the highest distance scores (SSD in this algorithm) may not be the best patches to fill in the hole and to generate visually plausible results due to the limitation of SSD metric, especially for complex scenes with different color information and structures. The algorithm sometimes can lead to false matching, in which the reported “best” patch with a high correlation score may have very distinctive textural or color information compared to the object patch. Under such circumstances, the artificial effects introduced by the false matching will degrade the quality of the result images significantly. Fortunately, spatial locality can be observed in most of the natural images, therefore, the visually plausible matching patches (in terms of color and texture similarity) tend to reside in the surrounding area of the candidate patch with high chances. By reducing the search area to a certain degree, we can reduce the possibility of false matching and therefore generate visually plausible result images.

#### 5.4 Optimizing Patch Size

The object removal algorithm is an iterative algorithm, in which one object patch is processed in each iteration. We need roughly  $wh/P^2$  iterations to finish the whole process. That said, if we increase patch size  $P$ , fewer iterations are needed to fill the object area which may lead to shorter processing time. Meanwhile, the complexity of the SSD computation ( $O(P^2)$ ) becomes higher for the patch matching, which tends to increase the processing time. Therefore, it is not straightforward to determine the impact of increasing patch size  $P$  to the overall complexity and performance.

We use Fig. 10 to help us analyze the overall computation complexity. First of all, we assume the search factor  $\alpha$  is defined as in the previous section. We also define the search area as  $SA$  as in (7).

Secondly, because the patch size is  $P$ , any candidate patch within  $(P - 1)$  pixels range surrounding the object area  $\Omega$  will partially overlap with the object area. We define this overlapping area as  $OA$ , whose area can be calculated as:

$$OA = (2(P - 1) + w) \cdot (2(P - 1) + h) - wh. \quad (8)$$

If the candidate patch lies outside the overlapping area,<sup>1</sup> the complexity of the SSD computation can be estimated as  $O(P^2)$ . When the candidate patch and the object area overlaps,<sup>2</sup> we only use the pixels  $I_q$  in the intersection of the candidate patch  $\Psi_q$  and source image  $\Phi$  ( $I_q \in \Psi_q \cap \Phi$ ) to perform the computation. We can estimate the computation complexity as  $O(kP^2)$ , in which  $k$  is a constant value, representing the average ratio of the overlapping area. For a typical rectangle search area and object area,  $k$  can be approximately estimated as 0.5. Therefore, the overall computation complexity can be calculated as in (9).

Experimental results show that the processing time can be reduced by increasing patch size while reducing the search area. From Fig. 10, an intuitive explanation is that when we reduce the search area, more candidate patches overlap with the object region. In this scenario, the bigger the patches are, the more overlap there will be. As a result, the amount of required computations becomes smaller. Thus, as we increase the patch size and reduce the search area, the processing time decreases. Equation (9) shows that for bigger search factor  $\alpha$ , the term  $(2\alpha + 1)^2$  dominates the complexity. In this case, the complexity change caused by the increased patch size is negligible. However, when  $\alpha$  becomes smaller, the search area decreases. When the search area  $SA$  becomes comparable to or even smaller than the object area  $\Omega$ , the term  $(2\alpha + 1)^2$  becomes less dominant. Therefore, for smaller search factor  $\alpha$ , increasing the patch size  $P$  can reduce the computation complexity. Experimental results shown in Fig. 11 verify the above analysis. In Fig. 11, the processing time for patch sizes of  $13 \times 13$  and  $17 \times 17$  is normalized by the processing time of the  $9 \times 9$  patch to show the performance speedup achieved by increasing the patch size. Experimental results show that, for bigger search areas ( $\alpha \geq 1$ ), patch size does not affect the performance. However, as the search area keeps

<sup>1</sup>In this case, the candidate patch is in the area of  $(SA - OA)$ .

<sup>2</sup>In this case, the candidate patch is in the area of  $OA$ .

decreasing ( $\alpha < 1$ ), bigger patch size leads to more significant time reduction. The experimental results in Fig. 11

also indicate that increasing the patch size works for the experiments

$$\begin{aligned}
 \text{Complexity}_{\text{overall}} &\approx O(wh/P^2) \cdot ((2\alpha + 1)^2 - 1)wh \cdot \left[ \text{Prob}(\Psi_q \in (SA - OA)) \cdot O(P^2) + \text{Prob}(\Psi_q \in OA) \cdot O(kP^2) \right] \\
 &= O \left[ ((2\alpha + 1)^2 - 1)w^2h^2 \cdot \left( \frac{SA - OA}{SA} + k \frac{OA}{SA} \right) \right] \\
 &= O \left[ ((2\alpha + 1)^2 - 1)w^2h^2 \cdot \left( \frac{(2\alpha + 1)^2 wh - (2(P - 1) + w)(2(P - 1) + h)}{(2\alpha + 1)^2 - 1)wh} \right. \right. \\
 &\quad \left. \left. + k \frac{(2(P - 1) + w)(2(P - 1) + h) - wh}{((2\alpha + 1)^2 - 1)wh} \right) \right] \\
 &= O \left[ wh \cdot ((2\alpha + 1)^2 wh - (2(P - 1) + w)(2(P - 1) + h) + k((2(P - 1) + w)(2(P - 1) + h) - wh)) \right] \\
 &= O \left[ wh \cdot (((2\alpha + 1)^2 - k)wh - (1 - k)(2(P - 1) + w)(2(P - 1) + h)) \right] \\
 &= O \left[ w^2h^2 \cdot ((2\alpha + 1)^2 - (1 - k)(2 \frac{(P - 1)}{w} + 1)(2 \frac{(P - 1)}{h} + 1) - k) \right]. \tag{9}
 \end{aligned}$$

with (or without) the memory optimization discussed in Section 5.5. In addition, Fig. 12 shows that by increasing the patch size, we can generate visually plausible result images without degrading the image quality.

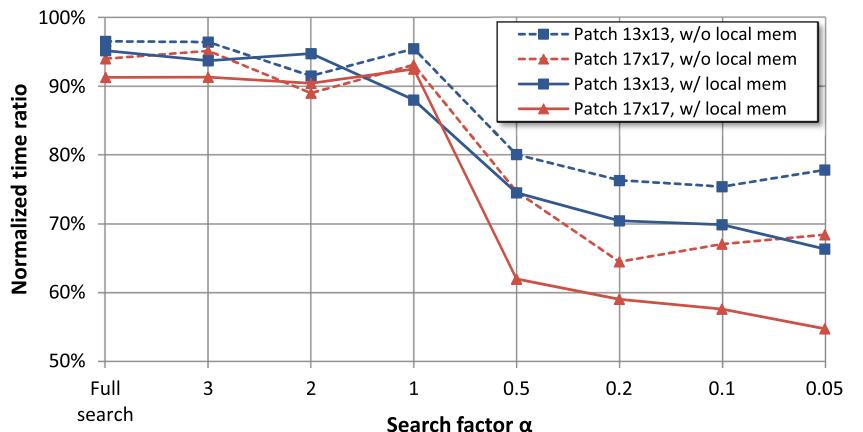
### 5.5 Memory Optimization

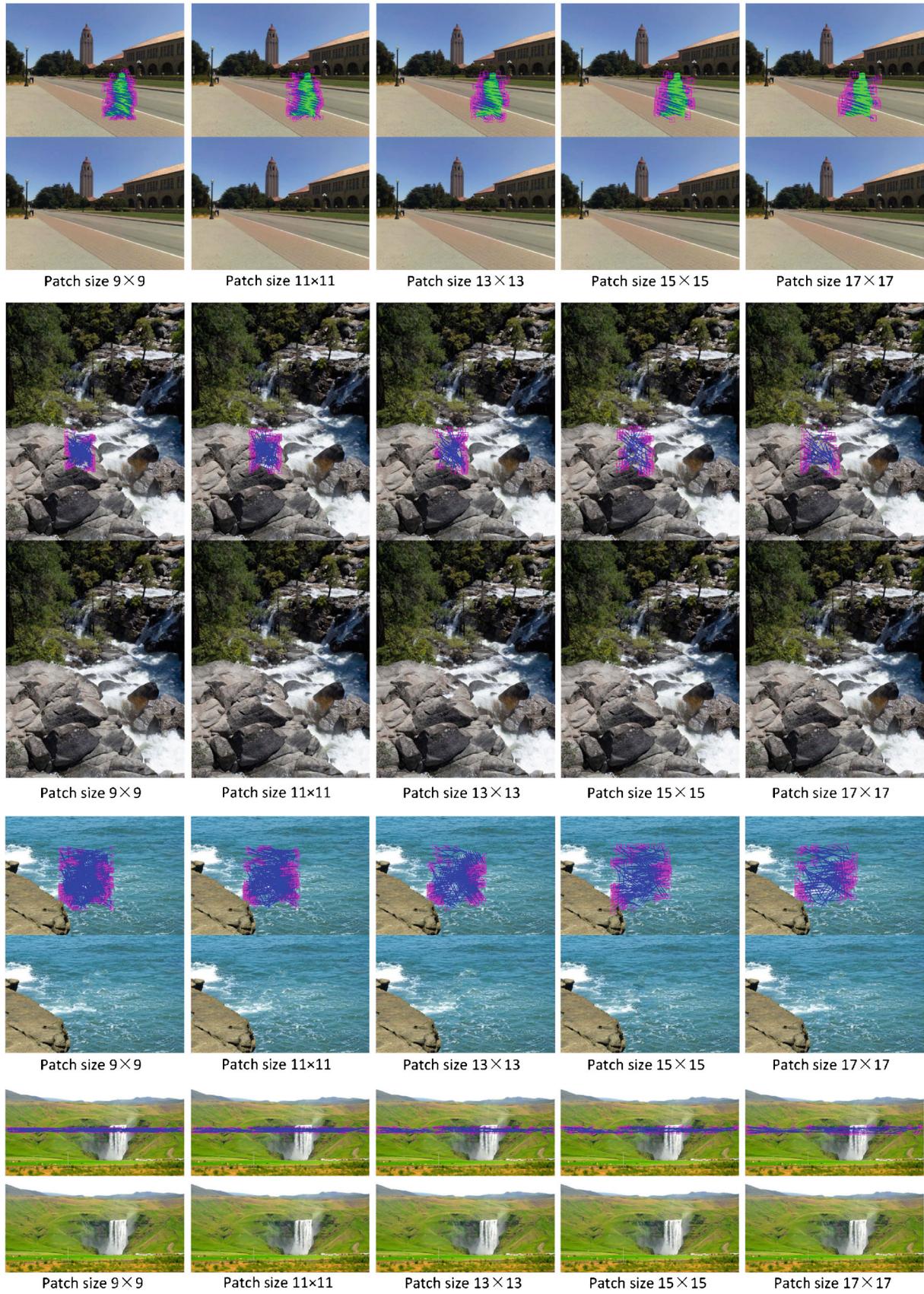
Similar to desktop GPUs, mobile GPUs also suffer from long latency of the off-chip global memory. The local memory on the mobile GPU provides fast memory accesses and can be shared by work items in the same work group. As mentioned before, a work group contains  $8 \times 8$  work items, each of which computes an SSD value between an object patch and a candidate patch. As shown in Fig. 13, the  $n$ -th work item works on the  $n$ -th patch. Most of the pixels in adjacent candidate patches overlap. For instance, patch 1 and patch 2 share most of the image pixels and

only one column of pixels in each patch are different. Similarly, all adjacent candidate patches processed by  $8 \times 8$  work items have many overlapped pixels, each of which is accessed multiple times by several different work items. These unnecessary memory accesses to the global memory can lead to long latency and increase the processing time. We can also tell from Fig. 13 that for a  $P \times P$  patch,  $(P + 8 - 1) \times (P + 8 - 1)$  pixels are actually shared among work items. Thus, we can load these pixels into the on-chip local memory to allow data sharing and avoid unnecessary accesses to the global memory. In our OpenCL implementation,  $(P + 8 - 1)^2 \cdot \text{sizeof}(\text{cl\_uchar4})$  source image data,  $P^2 \cdot \text{sizeof}(\text{cl\_uchar4})$  patch image data and  $P^2 \cdot \text{sizeof}(\text{cl\_int})$  patch pixel label data can be loaded into the local memory.

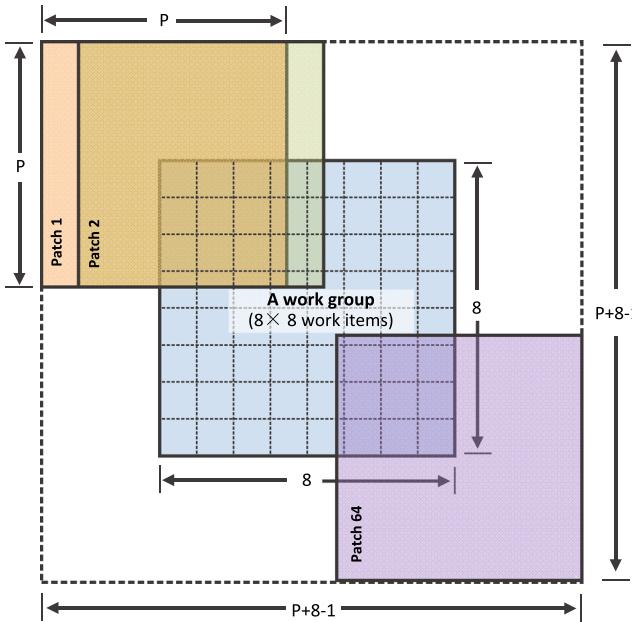
Table 4 lists the local memory usage for different patch sizes for the “WalkingMan” test image. The required local

**Figure 11** Impact of increased patch size  $13 \times 13$  and  $17 \times 17$ . “WalkingMan” test image. The processing time for  $13 \times 13$  and  $17 \times 17$  patches is normalized by the time of the  $9 \times 9$  patch. (The  $9 \times 9$  patch size is suggested by the original algorithm proposed by Criminisi et al. [4]).





**Figure 12** The experimental results for increased patch size. ( $\alpha = 0.05$  in this experiment).



**Figure 13** A detailed diagram showing how  $8 \times 8$  work items in a work group compute SSD values in parallel. Most of the pixels in the dashed box are accessed multiple times by parallel work items.

memory for all the patch sizes listed in Table 4 can be fit into the 8KB local memory of the Adreno GPU. In addition, if we carefully design the method to load data from the global memory to the local memory by data striping, we can coalesce the global memory access to further reduce latency. The parallel data loading from the global memory to the local memory is shown in Algorithm 2, in which the coalesced global memory access is achieved.

Experimental results in Fig. 14 demonstrate the performance improvement by utilizing the local memory to

**Figure 14** Performance comparison between using the local memory and without using the local memory. The results for the “WalkingMan” are shown. The experiments performed on other test images generate similar results.

enable the data sharing between work items inside the same work group. We observe on average a 30 % reduction in processing time after using the local memory.

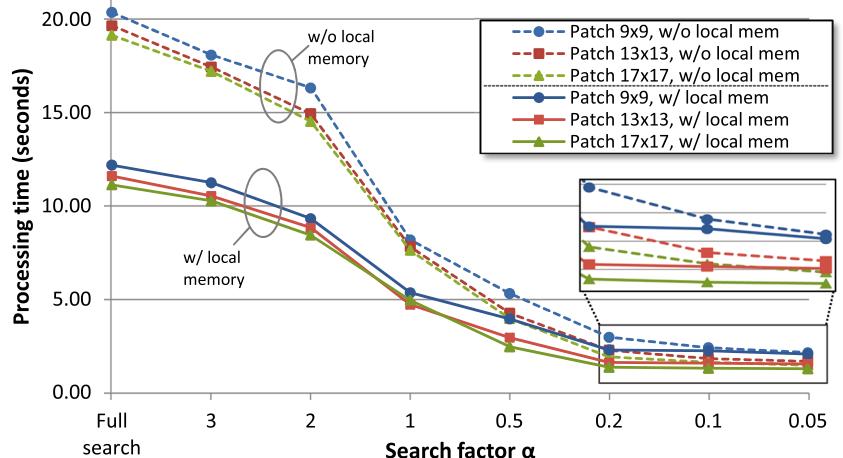
**Algorithm 2** Parallel data loading from global memory to local memory in *findBestPatch* OpenCL kernel function.

```

1. Get global ID of the current work item: (gid.x, gid.y);
2. Get local ID of the current work item: (lid.x, lid.y);
3. local_id = lid.y * lsize.x + lid.x;
4. Get local work group size: (lsize.x, lsize.y);
5. group_size = lsize.x * lsize.y;
6. local_mem_size = (P + 8 - 1) * (P + 8 - 1);
7. while(local_id < local_mem_size)
8.   if(local_id < P * P)
9.     Calculate global memory address for patchData and
    patchPixelLabel;
10.    Load patchPixelLabel into local memory;
11.    Load patchData into local memory;
12.   end if
13.   Calculate global memory address for srcData;
14.   Load srcData into local memory;
15.   local_id += group_size;
16. end while
17. barrier(CLK_LOCAL_MEM_FENCE);
18. Start SSD computation from here. (Similar to Algorithm 1,
just read needed data from local memory.)
```

## 6 Experimental Results

We implemented the exemplar-based inpainting algorithm for object removal on a test platform based on the Snapdragon chipset using OpenCL and the Android NDK [7, 13]. We applied the proposed optimization techniques





**Figure 15** An interactive object removal demo on Android with the OpenCL acceleration. **(a)** original image; **(b)** a mask indicating the object area; **(c)** intermediate result; **(d)** final result image after iterative editing.

discussed in Section 5. To demonstrate the efficiency and practicality of the proposed implementation, we developed an interactive OpenCL Android demo on the test platform. Figure 15 shows screen-shots of the implemented Android demo application, in which an end user can draw a customized mask by touching the touchscreen to cover an unwanted object and then remove it by pressing a button.

**Table 4** Local memory usage for “WalkingMan” image.

| Patch size                           | Data              | Local memory |
|--------------------------------------|-------------------|--------------|
| $9 \times 9$<br>$(p + 8 - 1 = 16)$   | Source data       | 1024 bytes   |
|                                      | Patch data        | 324 bytes    |
|                                      | Patch pixel label | 324 bytes    |
|                                      | Total             | 1672 bytes   |
| $13 \times 13$<br>$(p + 8 - 1 = 20)$ | Source data       | 1600 bytes   |
|                                      | Patch data        | 676 bytes    |
|                                      | Patch pixel label | 676 bytes    |
|                                      | Total             | 2952 bytes   |
| $17 \times 17$<br>$(p + 8 - 1 = 24)$ | Source data       | 2304 bytes   |
|                                      | Patch data        | 1156 bytes   |
|                                      | Patch pixel label | 1156 bytes   |
|                                      | Total             | 4616 bytes   |

The demo allows iterative editing, so that the user can keep editing an image until a satisfying result is obtained.

From Tables 5–8, we show the processing time of the OpenCL-based implementations, including the CPU-only results (utilizing multi-core CPU on the chip) and CPU-GPU heterogeneous implementations. We can notice that the OpenCL implementations with proposed optimizations significantly improve the processing performance compared to the serial version implementation. The CPU-GPU heterogeneous implementations further improve the performance compared to the CPU-only implementations.

Table 5 shows experimental results for the “Walking-Man” image, in which the image size is  $512 \times 384$ , and the size of the object area  $76 \times 128$ . The mask is manually drawn to cover the walking person. With the default parameter configuration ( $9 \times 9$  patch size, full search), the serial version of the implementation running on one CPU core uses 87 seconds to finish the processing (shown in Table 2), which is a long processing time for a practical mobile application. The fact that iterative editing is required under many circumstances makes the serial implementation far from being practical. Table 5 shows experimental results for OpenCL-based parallel solutions. With the default parameter configuration, the multi-core CPU-only version reduces

**Table 5** Total processing time for “WalkingMan” image, with OpenCL kernels running on the GPU.

| Search factor $\alpha$ | Search area | CPU-only   |         |         | Heterogeneous CPU+GPU     |         |         |                          |         |         |
|------------------------|-------------|------------|---------|---------|---------------------------|---------|---------|--------------------------|---------|---------|
|                        |             | Time       |         |         | Time w/o local memory (s) |         |         | Time w/ local memory (s) |         |         |
|                        |             | Patch size |         |         | Patch size                |         |         | Patch size               |         |         |
|                        |             | 9 × 9      | 13 × 13 | 17 × 17 | 9 × 9                     | 13 × 13 | 17 × 17 | 9 × 9                    | 13 × 13 | 17 × 17 |
| full search            | 512 × 384   | 23.26      | 22.48   | 23.68   | 20.37                     | 19.66   | 19.15   | 12.18                    | 11.59   | 11.12   |
| 2                      | 382 × 384   | 18.08      | 16.97   | 18.30   | 16.32                     | 14.93   | 14.52   | 9.31                     | 8.82    | 8.42    |
| 1                      | 234 × 311   | 13.37      | 9.91    | 9.74    | 8.17                      | 7.80    | 7.61    | 5.34                     | 4.69    | 4.94    |
| 0.5                    | 156 × 248   | 11.27      | 9.80    | 8.20    | 5.29                      | 4.24    | 3.95    | 3.93                     | 2.93    | 2.43    |
| 0.2                    | 109 × 176   | 7.13       | 5.20    | 3.79    | 2.95                      | 2.25    | 1.90    | 2.26                     | 1.59    | 1.33    |
| 0.05                   | 86 × 139    | 6.02       | 4.73    | 3.85    | 2.12                      | 1.65    | 1.45    | 2.04                     | 1.52    | 1.25    |

**Table 6** Total processing time for “River” image. With OpenCL kernels running on the GPU.

| Search factor $\alpha$ | Search area | CPU-only   |         |         | Heterogeneous CPU+GPU     |         |         |                          |         |         |
|------------------------|-------------|------------|---------|---------|---------------------------|---------|---------|--------------------------|---------|---------|
|                        |             | Time       |         |         | Time w/o local memory (s) |         |         | Time w/ local memory (s) |         |         |
|                        |             | Patch size |         |         | Patch size                |         |         | Patch size               |         |         |
|                        |             | 9 × 9      | 13 × 13 | 17 × 17 | 9 × 9                     | 13 × 13 | 17 × 17 | 9 × 9                    | 13 × 13 | 17 × 17 |
| full search            | 480 × 639   | 25.94      | 25.01   | 27.41   | 18.28                     | 16.19   | 16.01   | 14.77                    | 13.80   | 13.35   |
| 2                      | 295 × 465   | 12.23      | 12.15   | 12.68   | 8.52                      | 7.93    | 7.95    | 6.71                     | 6.94    | 6.59    |
| 1                      | 177 × 279   | 7.76       | 6.98    | 5.34    | 5.09                      | 3.13    | 3.11    | 4.15                     | 2.74    | 2.60    |
| 0.5                    | 118 × 186   | 4.80       | 3.91    | 3.82    | 2.81                      | 2.06    | 1.42    | 2.34                     | 1.87    | 1.5     |
| 0.2                    | 83 × 130    | 3.21       | 1.66    | 1.57    | 1.93                      | 1.18    | 1.05    | 1.84                     | 1.34    | 1.06    |
| 0.05                   | 65 × 102    | 2.29       | 1.72    | 1.91    | 1.39                      | 1.15    | 1.01    | 1.66                     | 1.16    | 1.00    |

**Table 7** Total processing time for “Dive” image. With OpenCL kernels running on the GPU.

| Search factor $\alpha$ | Search area | CPU-only   |         |         | Heterogeneous CPU+GPU     |         |         |                          |         |         |
|------------------------|-------------|------------|---------|---------|---------------------------|---------|---------|--------------------------|---------|---------|
|                        |             | Time       |         |         | Time w/o local memory (s) |         |         | Time w/ local memory (s) |         |         |
|                        |             | Patch size |         |         | Patch size                |         |         | Patch size               |         |         |
|                        |             | 9 × 9      | 13 × 13 | 17 × 17 | 9 × 9                     | 13 × 13 | 17 × 17 | 9 × 9                    | 13 × 13 | 17 × 17 |
| full search            | 576 × 384   | 64.48      | 62.86   | 61.47   | 44.83                     | 40.45   | 36.65   | 44.62                    | 34.97   | 31.08   |
| 2                      | 576 × 384   | 65.11      | 64.83   | 63.55   | 44.73                     | 40.29   | 36.61   | 44.67                    | 35.00   | 31.11   |
| 1                      | 465 × 384   | 52.17      | 54.31   | 54.99   | 34.22                     | 33.26   | 19.33   | 28.62                    | 28.65   | 27.81   |
| 0.5                    | 310 × 369   | 54.36      | 37.07   | 36.82   | 21.78                     | 22.73   | 8.24    | 19.98                    | 23.04   | 16.68   |
| 0.2                    | 217 × 260   | 40.92      | 29.64   | 27.69   | 15.14                     | 9.45    | 5.66    | 14.01                    | 10.40   | 7.40    |
| 0.05                   | 171 × 205   | 35.44      | 22.22   | 20.171  | 13.74                     | 7.90    | 5.40    | 14.21                    | 7.84    | 5.31    |

the processing time to 23.26 seconds, and the heterogeneous CPU-GPU implementation further reduces the processing time to 20.37 seconds (76.6 % time reduction compared to 87 seconds processing time for the serial implementation).

With all the proposed optimization techniques applied, we observe significant performance speedup. With search factor  $\alpha = 0.05$ , patch size  $17 \times 17$ , and local memory enabled, the processing time is reduced to only 1.25 seconds, which indicates a 93.9 % reduction in processing

**Table 8** Total processing time for “Hill” image. With OpenCL kernels running on the GPU.

| Search factor $\alpha$ | Search area | CPU-only   |         |         | Heterogeneous CPU+GPU     |         |         |                          |         |         |
|------------------------|-------------|------------|---------|---------|---------------------------|---------|---------|--------------------------|---------|---------|
|                        |             | Time       |         |         | Time w/o local memory (s) |         |         | Time w/ local memory (s) |         |         |
|                        |             | Patch size |         |         | Patch size                |         |         | Patch size               |         |         |
|                        |             | 9 × 9      | 13 × 13 | 17 × 17 | 9 × 9                     | 13 × 13 | 17 × 17 | 9 × 9                    | 13 × 13 | 17 × 17 |
| full search            | 1024 × 550  | 153.36     | 217.15  | 208.62  | 248.55                    | 329.53  | 313.33  | 94.61                    | 103.18  | 90.30   |
| 2                      | 1024 × 50   | 21.07      | 15.86   | 12.14   | 20.77                     | 21.49   | 20.88   | 18.9                     | 11.54   | 12.22   |
| 1                      | 1024 × 30   | 14.46      | 14.00   | 12.73   | 16.80                     | 11.41   | 12.73   | 14.7                     | 13.61   | 9.31    |
| 0.5                    | 1024 × 20   | 12.88      | 14.15   | 12.09   | 15.61                     | 12.09   | 12.67   | 15.9                     | 11.28   | 8.91    |
| 0.2                    | 1024 × 14   | 13.33      | 14.00   | 11.77   | 15.50                     | 14.22   | 12.84   | 16.2                     | 11.54   | 9.37    |
| 0.05                   | 1024 × 12   | 11.52      | 15.46   | 11.18   | 15.60                     | 17.11   | 13.38   | 15.9                     | 13.34   | 8.79    |

**Table 9** Speedup for OpenCL-based heterogeneous implementations with the proposed algorithmic and memory optimizations.

| Image      | Processing time (s) |                 |                  | Speedup |
|------------|---------------------|-----------------|------------------|---------|
|            | w/o opt.            | w/ opt.         |                  |         |
|            | Full search         | $\alpha = 0.05$ | Patch size 9 × 9 |         |
| WalkingMan | 20.37               | 1.25            | 16.3 X           |         |
| River      | 18.28               | 1.00            | 18.3 X           |         |
| DivingMan  | 44.83               | 5.31            | 8.44 X           |         |
| Hill       | 248.55              | 8.79            | 28.3 X           |         |

time compared to 20.37 seconds for the default parameter configuration (full search, 9 × 9 patch, without using the local memory). The subjective quality of resultant images does not degrade according to experimental results shown in Fig. 9 and 12. According to the research conducted by Niida et al., users of mobile applications can tolerate several seconds average processing time for mobile services before they start to feel frustrated [17]. By accelerating the object removal algorithm using heterogeneous CPU-GPU partitioning on mobile devices, we successfully reduce the run time, which makes these types of computer vision algorithms feasible in practical mobile applications.

We can draw similar conclusions from the timing results for other test images shown in Tables 6, 7 and 8. To demonstrate the effectiveness of our proposed optimization schemes, the speedup gained from the proposed optimization strategies are concluded in Table 9. We observe speedups from 8.44X to 28.3X with all our proposed optimizations applied to the heterogeneous OpenCL implementations.

## 7 Conclusions

The emerging heterogeneous architecture of mobile SoC processors with the support of parallel programming models such as OpenCL enables the capability of general-purpose computing on the mobile GPU. As a case study, we present an OpenCL-based mobile GPU implementation of an object removal algorithm. We analyze the workload for the computationally-intensive kernels of the algorithm, and partition the algorithm between the mobile CPU and GPU. Algorithm mapping methodology and optimization techniques for OpenCL-based parallel implementation are discussed. Several implementation trade-offs are discussed according to the architectural properties of the mobile GPU. We perform experiments on a real mobile platform powered by a Snapdragon mobile SoC. The experimental results show that the CPU-GPU heterogeneous implementation with the proposed optimization strategies can significantly reduce the processing time without degrading the visual image quality. When we apply the proposed optimizations (setting the search factor to  $\alpha = 0.05$  and increasing the patch size to 17 × 17), we observe speedups from 8.44X to 28.3X for different test images. With the rapid development of mobile SoCs with heterogeneous computing capability, we foresee that many more computer vision applications with high complexity will be enabled on real-world mobile devices.

**Acknowledgment** This work was supported in part by Qualcomm, and by the US National Science Foundation under grants CNS-1265332, ECCS-1232274, and EECs-0925942.

## References

1. Bordallo Lopez, M., Nykänen, H., Hannuksela, J., Silven, O., Vehviläinen, M. (2011). Accelerating image recognition on mobile devices using GPGPU. In *Proceeding of SPIE* (Vol. 7872, p. 78720R).
2. Cheng, K.T., & Wang, Y. (2011). Using mobile GPU for general-purpose computing - a case study of face recognition on smartphones. In *Proceedings of IEEE international symposium VLSI design, automation and test (VLSI-DAT)* (pp. 1–4). doi:[10.1109/VDAT.2011.5783575](https://doi.org/10.1109/VDAT.2011.5783575).
3. Clemons, J.L. ((2013)). *Computer architectures for mobile computer vision systems*. Ph.D. Thesis, University of Michigan. URL <http://hdl.handle.net/2027.42/97782>.
4. Criminisi, A., Perez, P., Toyama, K. (2003). In *Proceedings in IEEE conference computer vision and pattern recognition (CVPR)* (Vol. 2, pp. 721–728). doi:[10.1109/CVPR.2003.1211538](https://doi.org/10.1109/CVPR.2003.1211538).
5. Ensor, A., & Hall, S. (2011). GPU-based image analysis on mobile devices. arXiv:[1112.3110](https://arxiv.org/abs/1112.3110).
6. Fernández, V., Orduna, J.M., Morillo, P. (2012). Performance characterization of mobile phones in augmented reality marker tracking. In *International conference of computational and mathematical methods in science and engineering (CMMSE)* (pp. 537–549).
7. Google (2013). Android Development Guide. URL <http://developer.android.com/index.html>.
8. Hofmann, R. (2012). *Extraction of natural feature descriptors on mobile GPUs*. Thesis Hochschulschriftenserver der Universität Koblenz-Landau.
9. Hofmann, R., Seichter, H., Reitmayr, G. (2012). A GPGPU accelerated descriptor for mobile devices. In *IEEE international symposium mixed and augmented reality (ISMAR)* (pp. 289–290).
10. Limited (2013). Imagination Technologies, PowerVR Graphics. URL <http://www.imgtec.com/powervr/powervr-graphics.asp>.
11. Lee, S.E., Zhang, Y., Fang, Z., Srinivasan, S., Iyer, R., Newell, D. (2009). Accelerating mobile augmented reality on a handheld platform. In *Proceedings of IEEE international conference computer design (ICCD)* (pp. 419–426).
12. Leskela, J., Nikula, J., Salmela, M. (2009). OpenCL Embedded Profile prototype in mobile device. In *Proceedings in IEEE workshop signal processing system (SiPS)* (pp. 279–284). doi:[10.1109/SIPS.2009.5336267](https://doi.org/10.1109/SIPS.2009.5336267).
13. Munshi, A. (2010). The OpenCL Specification v1.1, the Khronos Group. URL <http://www.khronos.org/opencl>.
14. Munshi, A., & Leech, J. (2009). The OpenGL ES 2.0 Specification, the Khronos Group. URL <http://www.khronos.org/opengles>.
15. Munshi, A., Gaster, B., Mattson, T. G., Fung, J., Ginsburg, D. (2011). OpenCL programming guide.: Addison-Wesley.
16. Nah, J.H., Kang, Y.S., Lee, K.J., Lee, S.J., Han, T.D., Yang, S.B. (2010). MobiRT: an implementation of OpenGL ES-based CPU-GPU hybrid ray tracer for mobile devices. In *ACM SIGGRAPH ASIA* (p. 50).
17. Niida, S., Uemura, S., Nakamura, H. (2010). Mobile services - user tolerance for waiting time. *IEEE Vehicular Technology Magazine*, 5(3), 61–67. doi:[10.1109/MVT.2010.937850](https://doi.org/10.1109/MVT.2010.937850).
18. NVIDIA Corp (2013). CUDA toolkit v5.5. URL <https://developer.nvidia.com/cuda-toolkit>.
19. NVIDIA Corporation (2013). NVIDIA Tegra mobile processor. URL <http://www.nvidia.com/object/tegra.html>.
20. Paucher, R., & Turk, M. (2010). Location-based augmented reality on mobile phones. In *IEEE computer society conf. Computer vision and pattern recognition workshops (CVPRW)* (pp. 9–16).
21. Pulli, K., Baksheev, A., Konyakov, K., Eruhimov, V. (2012). Real-time computer vision with OpenCV. *Communications of the ACM*, 55(6), 61–69.
22. Qualcomm Inc (2013). Qualcomm Snapdragon Processor. URL <http://www.qualcomm.com/chipsets/snapdragon>.
23. Rister, B., Wang, G., Wu, M., Cavallaro, J.R. (2013). A fast and efficient SIFT detector using the mobile GPU. In *Proceedings of IEEE international conference acoustics, speech, and signal processing (ICASSP)* (pp. 2674–2678).
24. Singhal, N., Park, I.K., Cho, S. (2010). Implementation and optimization of image processing algorithms on handheld GPU. In *Proceedings of IEEE international conference image processing (ICIP)* (pp. 4481–4484). doi:[10.1109/ICIP.2010.5651740](https://doi.org/10.1109/ICIP.2010.5651740).
25. Wagner, D., Reitmayr, G., Mulloni, A., Drummond, T., Schmalstieg, D. (2008). Pose tracking from natural features on mobile phones. In *Proceedings of international symposium mixed and augmented reality (ISMAR)* (pp. 125–134).
26. Wang, G., Rister, B., Cavallaro, J.R. (2013a). Workload analysis and efficient OpenCL-based implementation of SIFT algorithm on a smartphone. In *Proceedings in IEEE global conference signal and information processing (GlobalSIP)* (pp. 759–762).
27. Wang, G., Xiong, Y., Yun, J., Cavallaro, J.R. (2013b). Accelerating computer vision algorithms using OpenCL framework on the mobile GPU - a case study. In *Proceedings of IEEE international conference acoustics, speech, and signal processing (ICASSP)* (pp. 2629–2633).
28. Wang, Y.C., & Cheng, K.T. (2012). Energy and performance characterization of mobile heterogeneous computing. In *IEEE workshop on signal processing systems (SiPS)* (pp. 312–317). IEEE.
29. Xiong, Y., Liu, D., Pulli, K. (2009). Effective gradient domain object editing on mobile devices. In *Proceedings of Asilomar conference signals, systems and computers (ASILOMAR)* (pp. 1256–1260). doi:[10.1109/ACSSC.2009.5469959](https://doi.org/10.1109/ACSSC.2009.5469959).
30. Yang, X., & Cheng, K.T. (2012a). Accelerating SURF detector on mobile devices, ACM Multimedia. URL <http://lbmedia.ece.ucsb.edu/resources/ref/acmmm12.pdf>.
31. Yang, X., & Cheng, K.T. (2012b). LDB: an ultra-fast feature for scalable Augmented Reality on mobile devices. In *Proceedings of IEEE international symposium mixed and augmented reality (ISMAR)* (pp. 49–57).



**Guohui Wang** received the B.S. degree in electrical engineering from Peking University, Beijing, China, and the M.S. degree in computer science from the Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China. Since 2008, he has been working towards the Ph.D. degree in Department of Electrical and Computer Engineering, Rice University, Houston, Texas. His research interests include mobile computing, digital signal processing, and VLSI architecture for wireless communication systems.



**Yingen Xiong** currently works on Multimedia R&D at Qualcomm. His research interest areas include GPGPU computing, computer vision, pattern recognition, and computational photography. Previously he worked at Nokia Research Center. Before that he was a research professor in Virginia Polytechnic Institute and State University and Wright State University. He received PhD degree from Nanjing University of Aeronautics and Astronautics.



**Jay Yun** received B.S. in Electrical Engineering from UCLA in 1990, and M.Eng and Ph.D in Electrical and Computer Engineering from Cornell University in 1997 and 2001, respectively. From 1992 to 1995, he was at Qualcomm developing reference designs for the base station controller in CDMA cellular systems. From 2000, he has been involved in a number of developments within multimedia realm in Qualcomm, including large-scale

video hardware for movie distribution, video preprocessors, 3D camera and Qualcomm's first GPU hardware architecture. He is currently a Principal Engineer/Manager in GPU hardware team, leading efforts to improve the hardware architecture for compute use cases such as camera and computer vision applications.



**Joseph R. Cavallaro** received the B.S. degree from the University of Pennsylvania, Philadelphia, Pa, in 1981, the M.S. degree from Princeton University, Princeton, NJ, in 1982, and the Ph.D. degree from Cornell University, Ithaca, NY, in 1988, all in electrical engineering. From 1981 to 1983, he was with AT&T Bell Laboratories, Holmdel, NJ. In 1988, he joined the faculty of Rice University, Houston, TX, where he is currently a Professor of

electrical and computer engineering. His research interests include computer arithmetic, and DSP, GPU, FPGA, and VLSI architectures for applications in wireless communications. During the 1996–1997 academic year, he served at the National Science Foundation as Director of the Prototyping Tools and Methodology Program. He was a Nokia Foundation Fellow and a Visiting Professor at the University of Oulu, Finland in 2005 and continues his affiliation there as an Adjunct Professor. He is currently the Director of the Center for Multimedia Communication at Rice University. He is a Senior Member of the IEEE and a Member of the IEEE SPS TC on Design and Implementation of Signal Processing Systems and the IEEE CAS TC on Circuits and Systems for Communications. He is currently an Associate Editor of the IEEE Transactions on Signal Processing, the IEEE Signal Processing Letters, and the Journal of Signal Processing Systems. He was Co-chair of the 2004 Signal Processing for Communications Symposium at the IEEE Global Communications Conference and General/Program Co-chair of the 2003, 2004, and 2011 IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP), General/Program Co-chair for the 2012, 2014 ACM/IEEE GLSVLSI, and Finance Chair for the 2013 IEEE GlobalSIP conference.