# ACCELERATING COMPUTER VISION ALGORITHMS USING OPENCL FRAMEWORK ON THE MOBILE GPU - A CASE STUDY

*Guohui Wang*, Yingen Xiong[†], Jay Yun[†], and Joseph R. Cavallaro**

*ECE Department, Rice University, Houston, Texas
[†]Qualcomm Technologies, Inc., California

## ABSTRACT

Recently, general-purpose computing on graphics processing units (GPGPU) has been enabled on mobile devices thanks to the emerging heterogeneous programming models such as OpenCL. The capability of GPGPU on mobile devices opens a new era for mobile computing and can enable many computationally demanding computer vision algorithms on mobile devices. As a case study, this paper proposes to accelerate an exemplar-based inpainting algorithm for object removal on a mobile GPU using OpenCL. We discuss the methodology of exploring the parallelism in the algorithm as well as several optimization techniques. Experimental results demonstrate that our optimization strategies for mobile GPUs have significantly reduced the processing time and make computationally intensive computer vision algorithms feasible for a mobile device. To the best of the authors' knowledge, this work is the first published implementation of general-purpose computing using OpenCL on mobile GPUs.

***Index Terms*—** GPGPU, mobile SoC, computer vision implementation, CPU-GPU algorithm partitioning, parallel architectures.

## 1. INTRODUCTION

Mobile computing technology has grown significantly over the past decade. As mobile processors are gaining more computing capability, we are witnessing a rapid growth of computer vision applications on mobile devices, such as image editing, augmented reality, object recognition and so on [1, 2].

One major usage of modern mobile devices is to take pictures and share them on social networks such as Facebook and Instagram. The demand for fast image editing functions on mobile devices has considerably increased. For example, users may want to remove unwanted objects in a picture taken by a camera phone before sharing it on the internet [3, 4]. However, long processing time due to the high computational complexity prevents computer vision algorithms from being practically used in mobile applications. To address this problem, researchers have explored the graphics processing units (GPUs) as accelerators to speedup image processing and computer vision algorithms [1, 5, 6, 7, 8, 9, 10, 11, 12]. Most of them use the OpenGL ES programming model to harness the compute power of the mobile GPU [13]. However, it is difficult to develop OpenGL ES programs for general-purpose computing, since the OpenGL ES model was originally designed for 3D graphics rendering on mobile platforms.

Recently, emerging programming models such as Open Computing Language (OpenCL) [14] and RenderScript [15] have been supported by mobile processors. As a result, GPGPU computing in the mobile domain becomes possible [1]. In this paper, we take the exemplar-based inpainting algorithm for object removal as a case study to explore the capability of mobile GPUs to accelerate computer vision algorithms using OpenCL. Our optimized GPU implementation shows significant speedup and enables fast interactive object removal applications in a practical mobile device.

The paper is organized as follows. Section 2 introduces the OpenCL programming model for mobile GPUs. Section 3 briefly explains the exemplar-based inpainting algorithm for object removal. We analyze the complexity of the algorithm and propose a method to map the algorithm onto a mobile GPU in Section 4. Section 5 shows experimental results on a practical mobile device. Section 6 concludes the paper.

## 2. OPENCL FOR MOBILE GPUS

Unlike the dedicated GPUs for desktop computers, a mobile GPU is typically integrated into an application processor, which also includes a multi-core CPU, an image processing engine, DSPs and other accelerators. Recently, modern mobile GPUs such as the Qualcomm Adreno GPU [16], the Imagination PowerVR GPU [17] and the NVIDIA ULP GeForce GPU [18] tend to integrate more compute units in a chip. Mobile GPUs have gained general-purpose parallel computing capability thanks to the multi-core architecture and emerging frameworks such as OpenCL. OpenCL is a programming framework designed for heterogeneous computing across various platforms [14]. 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

The first and fourth authors were supported in part by the US National Science Foundation under grants CNS-1265332, ECCS-1232274, EECS-0925942 and CNS-0923479.

978-1-4799-0356-6/13/$31.00 ©2013 IEEE

groups, and each of them consists of many work items which are the basic processing units to execute a kernel in parallel. OpenCL defines a hierarchical memory model containing a large global memory but with long latency and a small but fast 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 studied using OpenCL on mobile GPU. Leskela et al demonstrated an OpenCL Embedded Profile prototype emulated by OpenGL ES on mobile devices and showed advantages in performance and energy efficiency [19]. Since there is no previous work exploring OpenCL-based GPGPU computing on a real mobile device, it is desirable to study the GPGPU computing capability of a real mobile platform using the OpenCL framework and to explore the implementation and optimization methodology.

## 3. ALGORITHM OVERVIEW

Object removal is one of the most important image editing functions. 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 [3, 4]. In the meanwhile, this algorithm can achieve computational efficiency thanks to the block-based sampling processing, which is especially attractive for a parallel implementation.

Assume we have a source image $\Phi$ with a target region $\Omega$ to be filled in after an object is removed. We first search the image region $\Phi$ and find a patch $\Psi_{\tilde{q}}$ that best matches a patch $\Psi_p$ on the border of the target region $\Omega$. Then 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. Criminisi et al proposed a priority-based selection scheme to determine the patch filling order [3]. A patch priority is composed of two terms: a data term indicating the reliability of the surrounding pixels and a confidence term indicating the nearby structural information. The patch priority is evaluated for every pixel on the border of the target region $\Omega$. The patch to be filled in is chosen based on priority values. More details can be found in reference [3].

## 4. IMPLEMENTATION AND OPTIMIZATION

### 4.1. Mapping Object Removal Algorithm onto GPGPU

Fig. 1 shows a work flow diagram of the the exemplar-based inpainting algorithm for object removal. The blocks with the dashed lines are core functions inside the iterative loop and represent most of the computational workload. We can map the core functions into OpenCL kernels to exploit the 2-dimentional pixel-level and block-level parallelisms in the algorithms. The CPU handles the OpenCL context initialization and maintenance, memory objects management and kernel launching. By analyzing the algorithm, we partition the core functions into eight OpenCL kernels based on the properties of computations, as is shown in Table 1. 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 RGBA color pixel values, we use efficient vector data structures such as *cl_uchar4* to take advantage of built-in vector functions.

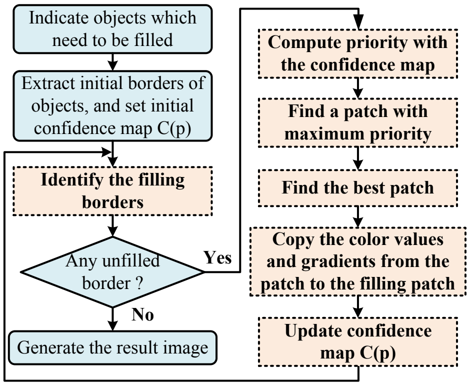

Indicate objects which need to be filled
Extract initial borders of objects, and set initial confidence map C(p)
Identify the filling borders
Any unfilled border ?
Yes
No
Generate the result image
Compute priority with the confidence map
Find a patch with maximum priority
Find the best patch
Copy the color values and gradients from the patch to the filling patch
Update confidence map C(p)

**Fig. 1**. Algorithm work flow diagram. The blocks with dashed lines are mapped into multiple OpenCL kernels. The other blocks are implemented using standard C.

**Table 1**. Breakdown of exec cycles for OpenCL kernels.

| Kernel functions | Exec cycle percentage |
|---|---|
| Convert RGB image to gray-scale image | 0.05% |
| Update the border of the area to be filled | 0.08% |
| Mark the source pixels to be used to fill the hole | 1.16% |
| Update pixel priorities in the filling area | 0.45% |
| Update pixel confidence in the filling area | 0.03% |
| **Find the best matching patch** | **98.20%** |
| Update the RGB image of the filling patch | 0.02% |
| Update the grayscale image of the filling patch | 0.02% |

To better optimize the OpenCL-based implementation, we first measure the performance of the OpenCL kernels using an internal performance profiler. Table 1 shows a breakdown of execution cycles on a Qualcomm Snapdragon S4 chipset [16]. The OpenCL kernel function used to find the best matching patch with the current patch ( *findBestPatch*) occupies most of the processing time (98.2%), so the optimization of this kernel is the key to improving performance.

### 4.2. Optimizations of GPU Implementation

The core mission of *findBestPatch* kernel is to find the best matching patch $\Psi_{\tilde{q}}$ from candidate patches $\Psi_q$ in the source image region $\Phi$, to match a object patch $\Psi_p$ in the object region $\Omega$ based on certain distance metric. The sum of

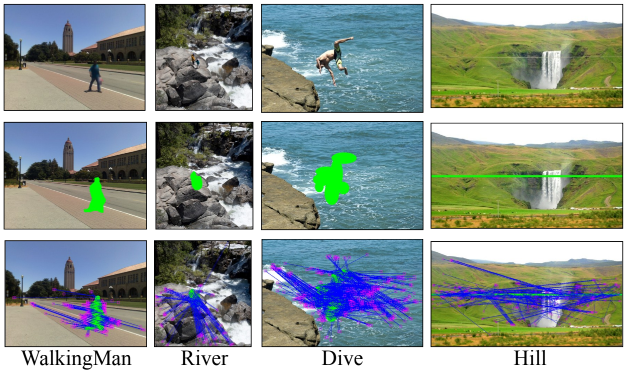


**Fig. 2**. The best patch mapping found by a full search. The 1st row: original images. The 2nd row: masks covering the unwanted objects. The 3rd row: best patch mapping.

squared differences (SSD) is used as a distance metric to measure the similarity between the patches [3]. 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 $\Psi_{\tilde{q}} = \arg\min_{q\in\Phi} d(\Psi_p, \Psi_q)$, in which $d(\Psi_q, \Psi_p) = \sum_{p\in\Psi_p\cap\Phi, q\in\Psi_q\cap\Phi}(I_p - I_q)^2$. Assume the size of the original image is $M \times N$, the size of the object area is $M_o \times N_o$, and the size of the patch is $P \times P$. The complexity of *findBestPatch* is $O(MNP^2)$. To perform a full search in the *findBestPatch* OpenCL kernel, we spawn $M \times N$ work items, with each computing an SSD value. We further partition these $M \times N$ work items into work groups according to the compute capability of the GPU. In our implementation on mobile GPU, each work group contains $8 \times 8$ work items.

### *4.2.1. Reducing search space of the best patch*

We have done an experiment to verify the locations of the best patches found by a full search across the whole image area. As is shown in Fig. 2, most of the best patches are found near the object area. The reason is that adjacent areas usually have the similar structures and textures in natural images. To reduce the searching time, we can utilize this spatial locality by limiting the search space.

In addition to the time reduction, reducing the search area has another benefit. 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. By limiting the search area, we can reduce the possibility of false matching, in which the “best” patch with a high correlation score may have very distinctive textural or color information compared to the object patch. Experiments show that reducing the search area by a certain degree can generate visually plausible result images.

We define the new search area by expanding the object area by $\alpha M_o$ to the up and down directions, and $\alpha N_o$ to the left and right directions. The search area factor $\alpha$ has a range $0 \leq \alpha < \max(M/M_o, N/N_o)$. The new search area becomes an area of $(M_o + 2\alpha M_o) \times (N_o + 2\alpha N_o)$. By defining the search factor, 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. The timing results for this optimization will be shown in Table 2 in Section 5.

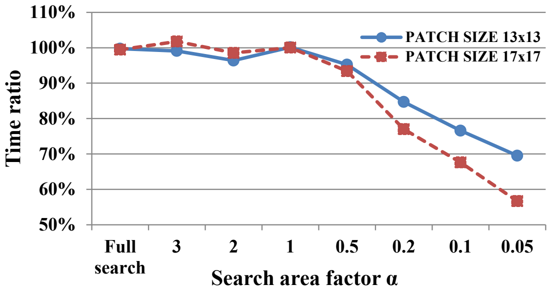


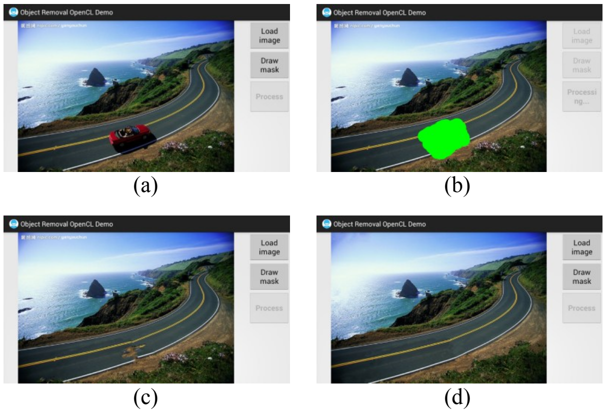
**Fig. 3**. Impact of increased patch size $13 \times 13$ and $17 \times 17$, compared to $9 \times 9$ patch. ($9 \times 9$ is the standard patch size suggested by the original algorithm [3].)

### *4.2.2. The impact of patch size*

The object removal algorithm is an iterative algorithm, in which an object patch is processed in an iteration. We need $M_oN_o/P^2$ iterations to finish the whole process. The overall complexity can be estimated as $O(MNP^2)\cdot O(M_oN_o/P^2) = O(MNM_oN_o)$, which seems to indicate that the patch size does not affect the processing time. However, the truth is that the patch size has a huge impact on the performance when we reduce the search area. 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 compute the SSD values. As we reduce the search area and increase the patch size, more candidate patches $\Psi_q$ overlap the object region $\Omega$. As a result, the area of $\Psi_q \cap \Phi$ becomes small, indicating a decrease in the number of pixels $I_q$ associated with computations. Therefore, the processing time is reduced. Experimental results shown in Fig. 3 verifies the above analysis. For bigger search areas ($\alpha \geq 1$), patch size does not affect the performance. However, as the search area keeps decreasing ($\alpha < 1$), bigger patch area leads to more significant time reduction.

### *4.2.3. Memory optimization*

Similar to desktop GPUs, mobile GPUs also suffer from long latency of the off-chip global memory. The local memory on the Adreno Mobile GPU provides quick 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. Adjacent candidate patches processed by these $8 \times 8$ work items have many overlapped pixels, each of which is accessed by several different work items. For a $P \times P$ patch, $(P + 8 - 1) \times (P + 8 - 1)$ pixels are shared. We can load these pixels into the local memory to allow data sharing and reduce the global memory accesses. In our OpenCL



(a) (b)

(c) (d)

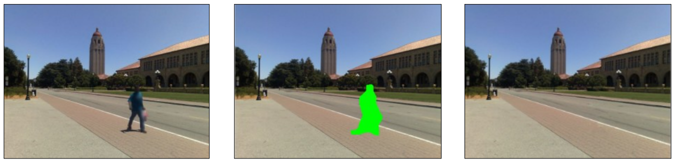
**Fig. 4**. Android demo with OpenCL acceleration. (a) original image; (b) a mask indicating the object area; (c) intermediate result; (d) final result image after iterative editing.

implementation, $(P+8-1)^2 \cdot sizeof(cl\_uchar4)$ source image data, $P^2 \cdot sizeof(cl\_uchar4)$ patch image data and $P^2 \cdot sizeof(cl\_int)$ patch pixel label data can be loaded into the local memory. For a $9\times 9$ patch ($P=9$), we need 1.63KB of local memory, which can be fit into the 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.

## 5. EXPERIMENTAL RESULTS

The Qualcomm Snapdragon S4 chipset contains a multi-core Krait CPU and Adreno GPU, and it supports the OpenCL Embedded Profile for both CPU and GPU [16]. We implemented the exemplar-based inpainting algorithm for object removal on a test platform based on the Snapdragon S4 chipset using OpenCL and the Android NDK [14, 15]. We applied the proposed optimization techniques. To demonstrate the efficiency and practicality of the proposed implementation, we developed an interactive OpenCL Android demo on the test platform. Fig. 4 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. The demo allows iterative editing, so that the user can keep editing an image until a satisfying result is obtained.

Due to the paper page limits, we only show experimental results for the "WalkingMan" image shown in Fig. 5. The size of the image is $512\times 384$, and the size of the object area is $76\times 128$. The mask is manually drawn to cover the walking person. When running on the CPU only, the OpenCL program uses 393.8 seconds, which is a long processing time for a practical mobile application. The fact that iterative editing is required under many circumstances makes the CPU-only implementation far from being practical. Table 2 shows experimental results for the CPU-GPU hetergeneous solution. We study the impact of reduced search areas and increased patch sizes. On one hand, given a fixed patch size, reducing the search area significantly decreases the run time. On the other hand, increasing the patch size for larger search areas (e.g. full search, $\alpha=2$, $\alpha=1$) does not affect the run time. However, for smaller search area (e.g. $\alpha=0.5, 0.2, 0.05$), we observe performance gains with larger patches. Experimental results justifies our analysis in Section 4.2.2.



Original image Mask image Result image

**Fig. 5**. Image "WalkingMan" used in the experiments.

**Table 2**. Total processing time for hetergeneous solution, with OpenCL kernels running on the GPU.

| Search factor | Search area | Time for different patch sizes (*seconds*) | | |
|---|---|---|---|---|
| $\alpha$ | | $9\times 9$ | $13\times 13$ | $17\times 17$ |
| full search | $512\times 384$ | 96.652 | 96.394 | 96.110 |
| 2 | $481\times 384$ | 73.367 | 70.723 | 72.296 |
| 1 | $234\times 311$ | 37.107 | 37.156 | 37.114 |
| 0.5 | $156\times 248$ | 19.975 | 19.024 | 18.658 |
| 0.2 | $108\times 176$ | 10.539 | 8.925 | 8.117 |
| 0.05 | $84\times 138$ | 7.528 | 5.233 | 4.266 |

With search factor $\alpha = 0.05$ and patch size $17\times 17$, the processing time is only 4.266 seconds, which indicates a 95.6% reduction in processing time compared with the "full search and $9\times 9$ patch" case. The subjective quality of resultant images does not degrade according to our experiment results. The subsequent experiments show that the processing time can be further reduced to less than 2 seconds on a high end Snapdragon S4 processor. According to the research conducted by Niida et al, users can tolerate 5 seconds average processing time for mobile services before they start to feel frustrated [20]. By accelerating the object removal algorithm using mobile GPU, we successfully reduce the run time to $1\sim 5$ seconds, which makes these types of computer vision algorithms feasible in practical mobile applications.

## 6. CONCLUSIONS

Modern mobile GPUs are capable of performing general-purpose computing with the support of programming models such as OpenCL. As a case study, we present an OpenCL-based mobile GPU implementation of an object removal algorithm. Algorithm mapping and optimization techniques for mobile GPUs are discussed. The experimental results on a real mobile platform powered by a Snapdragon S4 processor show that by offloading the core computations to mobile GPUs, the processing time can be significantly reduced. Therefore, we conclude that with the GPGPU support, many more computer vision applications can be enabled on practical mobile devices.

## 7. REFERENCES

[1] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, “Real-time computer vision with OpenCV,” *Communications of the ACM*, vol. 55, no. 6, pp. 61–69, 2012.

[2] X. Yang and K.-T. Cheng, “LDB: An ultra-fast feature for scalable Augmented Reality on mobile devices,” in *IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, 2012, pp. 49–57.

[3] A. Criminisi, P. Perez, and K. Toyama, “Object removal by exemplar-based inpainting,” in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 2, June 2003, pp. 721–728 vol.2.

[4] Y. Xiong, D. Liu, and K. Pulli, “Effective gradient domain object editing on mobile devices,” in *Asilomar Conference on Signals, Systems and Computers*, Nov. 2009, pp. 1256 –1260.

[5] S. E. Lee, Y. Zhang, Z. Fang, S. Srinivasan, R. Iyer, and D. Newell, “Accelerating mobile augmented reality on a handheld platform,” in *IEEE International Conference on Computer design (ICCD)*, 2009, pp. 419–426.

[6] H. Xie, K. Gao, Y. Zhang, J. Li, and Y. Liu, “GPU-based fast scale invariant interest point detector,” in *IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP)*, March 2010, pp. 2494 –2497.

[7] N. Singhal, I. K. Park, and S. Cho, “Implementation and optimization of image processing algorithms on handheld GPU,” in *IEEE International Conference on Image Processing (ICIP)*, Sept. 2010, pp. 4481 –4484.

[8] J.-H. Nah, Y.-S. Kang, K.-J. Lee, S.-J. Lee, T.-D. Han, and S.-B. Yang, “MobiRT: an implementation of OpenGL ES-based CPU-GPU hybrid ray tracer for mobile devices,” in *ACM SIGGRAPH ASIA 2010 Sketches*, 2010, p. 50.

[9] K.-T. Cheng and Y. Wang, “Using mobile GPU for general-purpose computing - a case study of face recognition on smartphones,” in *International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, April 2011, pp. 1 –4.

[10] M. Bordallo Lopez, H. Nykänen, J. Hannuksela, O. Silven, and M. Vehviläinen, “Accelerating image recognition on mobile devices using GPGPU,” in *Proceedings of SPIE*, vol. 7872, 2011, p. 78720R.

[11] X. Yang and K.-T. Cheng, “Accelerating SURF detector on mobile devices,” *ACM Multimedia*, Oct. 2012. [Online]. Available: http://lbmedia.ece.ucsb.edu/resources/ref/acmmm12.pdf

[12] B. Rister, G. Wang, M. Wu, and J. R. Cavallaro, “A fast and efficient SIFT detector using the mobile GPU,” in *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP)*, May 2013.

[13] Aaftab Munshi and Jon Leech, the Khronos Group, *The OpenGL ES 2.0 Specification*. [Online]. Available: http://www.khronos.org/opengles

[14] Aaftab Munshi, the Khronos Group, *The OpenCL Specification*. [Online]. Available: http://www.khronos.org/opencl

[15] Google Inc., *Android Development Guide*, http://developer.android.com/index.html.

[16] Qualcomm Inc., *Qualcomm Snapdragon Processor*, http://www.qualcomm.com/chipsets/snapdragon.

[17] Imagination Technologies Limited, *PowerVR Graphics*, http://www.imgtec.com/powervr/powervr-graphics.asp.

[18] NVIDIA Corpration, *NVIDIA Tegra mobile processor*, http://www.nvidia.com/object/tegra.html.

[19] J. Leskela, J. Nikula, and M. Salmela, “OpenCL Embedded Profile prototype in mobile device,” in *IEEE Workshop on Signal Processing Systems (SiPS)*, Oct. 2009, pp. 279 –284.

[20] S. Niida, S. Uemura, and H. Nakamura, “Mobile services - user tolerance for waiting time,” *IEEE Vehicular Technology Magazine*, vol. 5, no. 3, pp. 61 –67, Sept. 2010.