Real-time Integrated Face Detection and Recognition on Embedded GPGPUs

Saehanseul Yi, Illo Yoon, Chanyoung Oh, Youngmin Yi

School of Electrical and Computer Engineering, University of Seoul, Seoul, Korea
{shs.yi, yildbs, alspace11, ymyi}@uos.ac.kr

ABSTRACT
Both face detection and face recognition have started to be used widely these days in various applications such as biometric, surveillance, security, advertisement, entertainment, and so on. The ever increasing input image size in face detection and the large input DB in face recognition keep requiring more computational power to achieve real-time processing. Recently, embedded GPUs have started to support OpenCL and many applications can be accelerated successfully as the server GPUs have. In this paper, we propose several optimization techniques for the Local Binary Pattern (LBP) based integrated face detection and recognition algorithms, and successfully accelerated them achieving 22 fps using OpenCL on ARM Mali GPU, and 38 fps using CUDA on Tegra K1 GPU for HD inputs. This corresponds to 2.9 times and 3.7 times speedups respectively. To the best of our knowledge, it is the first paper that presents the acceleration of the face detection on embedded GPGPUs, and also that presents the performance of Tegra K1 GPU.

Keywords
Face Detection, Face Recognition, Embedded GPGPU, Mali, Tegra K1

1. INTRODUCTION
Face detection finds a face in a given image frame and is usually the first step in many applications which deal with the facial images. Face recognition, on the contrary, identifies the detected or any given test facial images, comparing with the ones in the DB. Both face detection and face recognition have started to be used widely these days in many areas: biometric and security for the building and door access, intelligent surveillance, commercials customization considering the age of the potential customers, HCI, drowsy driver detection system in an automobile, and so forth. In particular, CCTV cameras are rapidly deployed in more locations, but monitoring the constantly increasing number of videos by just human beings makes the system less effective. There is a strong need for automatic face detection and recognition in such a system.

As the resolution of the input image gets larger in face detection, more computational power is required for the real-time processing. For example, the input image in recent CCTV cameras is the size of full-HD, instead of HD or SD. In near future, UHD will be prevalent and more computation will be required for the real-time detection in such a large image. Likewise, as the number of images in the DB increases, the face recognition takes linearly more time.

On the other hand, graphics processing units (GPUs) have gained much popularity as it became general-purpose and one can offload computations to the GPU, where it would execute faster than on the CPU in most cases, as long as the application has high degree of parallelism. Thanks to such GPGPUs, CUDA and OpenCL based accelerated applications have proliferated over the past several years.

In addition to OpenCL supported embedded GPGPUs, the first CUDA-enabled embedded GPGPU called Tegra K1 has arrived very recently. It is reported that it contains 192 CUDA cores but it is known that each CUDA core is less powerful than a core in Mali or Adreno GPU. It was of our interest how much performance gain can be achieved on Tegra K1 GPU for the face detection and recognition application.

It is recently that the embedded GPUs, which reside inside an Application Processor (AP) in smart phones and tablets, have also become a GPGPU. Adreno, Mali, PowerVR, and many other modern GPUs started to support OpenCL, by which applications in various domains can be run on top of the GPU. Although the embedded GPUs have much less number of processing cores compared to server GPUs which have thousands of processing cores, equipped with a large vector engine it is quite powerful than the CPU, and the on-chip memory shared with the CPU alleviates the communication overheads between the CPU and the GPU, which is potentially one of the major performance bottlenecks in the CPU/GPU heterogeneous systems.

In this paper, we propose several optimization techniques for Local Binary Pattern (LBP) based face detection and recognition on embedded GPGPUs. We have successfully accelerated them using OpenCL on ARM Mali T604 GPU and also using CUDA on NVIDIA Tegra K1 GPU. Compared against the ARM Cortex-A15 single-threaded implementation with HD inputs, our face detection and recognition achieves 22 fps and 2.9 times speedup on Mali GPU, and 38 fps and 3.7 times speedup on Tegra K1 GPU. We analyze and discuss the effect of each optimization with extensive experiments.

There are quite a lot of work that accelerate face detection using GPGPU. However, to the best of our knowledge, it is the first paper that accelerates both the face detection and the face recognition on embedded GPGPUs. Also, we believe that it is the first paper that reports the performance of NVIDIA Tegra K1 GPU. It should be noted that simply running the CUDA or OpenCL implementations for server GPUs on embedded GPUs does not achieve the optimal performance. We confirmed it by running the CUDA face detection included in OpenCV library on Tegra K1 GPU. In
contrast, our proposed optimizations could achieve much better performance.

The remaining parts of the paper are organized as follows. Section 2 reviews the related work and Section 3 introduces some background of LBP based face detection and recognition. Section 4 explains the proposed technique, and the experimental results are shown in Section 5, followed by a conclusion in Section 6.

2. RELATED WORK

There has been a lot of work on acceleration of face detection using server GPUs. Sharma et al. proposed CUDA parallelization of Haar-based cascade face detection as well as face tracking on a GTX285 [3] and achieved more than 20 times speedup for VGA inputs. Oro et al. also proposed Haar-based face detection for HD video sequences on GTX470 [4][5] and achieved 2.5 times speedup on average. Hefenbrock et al. presented Haar-based cascade face detection using multiple GPUs and achieved the near best FPGA performance [6]. Wang et al. proposed a Haar-based face detection that both utilizes the CPU and the GPU inside a Sandy Bridge processor [7] and achieved 3.6 times speedup for VGA inputs. Face detection using Haar features involves an integral image method which is implemented using a prefix scan followed by a matrix transposition. Those works that accelerated Haar-based face detection tackles this compute-intensive part. However, an LBP-based approach has replaced Haar-based approaches as the LBP features are more discriminative and efficient in computation than the Haar features. As a result, OpenCV library has included MB-LBP based face detection using CUDA. MB-LBP features are basically similar to LBP features but known to be more discriminative than LBP ones by 8%, and also than Haar-like ones by 15% [15]. Note that the library also has an OpenCL face detection API but it uses Haar-like features, not LBP features. A more detailed comparison of an LBP-based face detector in OpenCV and our proposed face detector will be given in Section 5.

Accelerating face recognition on GPGPUs has been also attempted but there exists not much work compared to the face detection on GPGPUs. Ouerhani et al. presented a correlation based face recognition using CUDA on GeForce 8400 [8] and more than 2 times speedup was estimated. Dwith et al. presented LBP-based face recognition using OpenCL on Radeon HD 6500 [9]. They parallelized LBP algorithm achieving 30 times speedup but the Chi squared distance calculation, the performance bottleneck in the face recognition, was not parallelized whereas we deal with it in detail. Woo et al. proposed a PCA-based face recognition using CUDA on GTX580 [10] and achieved 30 times speedup.

On the other hand, the embedded GPUs have started to support OpenCL framework very recently. Wang et al. have accelerated an image removal algorithm on a Snapdragon S4 using OpenCL [13]. Both Adreno GPU and Krait CPU inside the chip run OpenCL kernels, and achieves 4 times speedup. Maghazeh et al. [12] compared the performance and the energy consumption ratio of Vivante GC2000 GPU to ARM Cortex-A9 quad-core CPU, with five benchmarks using OpenCL. We have also recently presented an LBP-based face recognition using OpenCL on Mali T604 GPU [11], where training modules focusing on LBP were parallelized. In contrast, in this paper, the testing modules in the face recognition has been parallelized focusing on Chi-Distance with different mappings and synchronizations.

When OpenCL was not supported before, Cheng et al. [16] present a case study of accelerating a Gabor-based face recognition system using OpenGL ES on a Tegra SoC which has dual-core ARM Cortex-A9 CPU and ULP GeForce GPU. The time-consuming face feature extraction task, which computes FFT based Gabor wavelet, was accelerated on the GPU, and achieved a 4.25 times speedup. Since OpenCL was not available, OpenGL ES was used in an ad-hoc way, which utilizes the GPU differently from OpenCL.

3. BACKGROUND

3.1 Mali GPU and Tegra K1 GPU

Mali T604 GPU is the first ARM GPU that supports OpenCL 1.1 Full Profile, and is capable of general purpose computing. It is a quad-core GPU, where each core runs at 533MHz and has two ALUs, and each ALU has 128-bit vector engine. On the other hand, NVIDIA has very recently released the first CUDA capable embedded GPU, Tegra K1. It has 192 CUDA cores which runs at from 400MHz to 852MHz.

The major difference between server and embedded GPUs is that the embedded GPUs have much less number of cores compared to server GPUs, and that embedded GPU is integrated with a CPU in a single chip. Inside the chip, CPU and GPU share the same physical memory, providing the unified memory where the host and the device can read and write data without copying them back and forth. The data transferring overhead between the CPU memory and the GPU memory can be significantly reduced.

Also among the embedded GPUs, there exist quite a large architectural differences. For example, each HW thread in a Mali has an independent program counter while Tegra K1 has a single program counter per each group of threads. More precisely, a group of threads, which is called warp, executes the same instruction all together on Tegra K1 architecture. Hence, if some of the threads in a warp branches, the other threads in the warp are halted until the branched threads return to the converging instruction stream. As a result, the divergent branch delays the execution of the warp. In constrast, as Mali has an individual program counter per thread, the warp size can be thought of as one, which makes Mali much more robust to divergent branches than Tegra K1. However, Tegra K1 integrated more ALUs in a chip instead of program counters, resulting in a larger computing power.

In addition, Mali does not provide fast shared memory (i.e., local memory in OpenCL term) where a work-group can share data with low latency, while Tegra K1 has the shared memory for a thread block. As we described already,
each core in a Mali has a vector unit and can be utilized with OpenCL vector operations while Tegra K1 does not. Considering these different architectural features, we optimized the same application with different optimization techniques for each target platform.

### 3.2 LBP-based Face Detection and Recognition

#### 3.2.1 Face Detection

The pipeline of LBP based face detection is shown in Figure 2. Among those steps, three dominant steps are Resizing, LBP, and Scanning. Figure 1 shows the profiling results measured with the CPU version of OpenCV implementation when skipping optimization was applied. Scanning is the largest performance bottleneck, followed by LBP and Resizing.

![Figure 1. Profiling result of LBP based face detection on CPU](image1.png)

First, the input image is resized into multiple smaller images according to the *scale factor*. Since the size of the face in the image is unknown, the search should be made with different size of a patch, meaning different face size. Instead of resizing the patch, for better efficiency, the patch size is fixed and the input image itself is resized as shown in Figure 4(a).

Then, LBP operation is applied to each of the resized images. Every pixel in the input image is compared with the neighboring pixels and it forms a binary pattern depending on the result of the comparison, which represents the textual features. Figure 3 shows the example of the LBP operation where each of eight neighboring pixels is compared with the center pixel (54) and is assigned either 0 or 1 depending on whether its value is greater than the center pixel. Then, the binary pattern (203) is produced by concatenating 0s and 1s, in a counter-clockwise manner in this example.

![Figure 2. The pipeline of LBP-based face detection algorithm](image2.png)

![Figure 3. LBP operation example](image3.png)

Once the image has transformed into LBP domain, the patch slides through all the search points, comparing the LBP feature values in the current patch with the trained values, in order to classify whether the window contains a face or not (Figure 4(b)). The classifier is called cascade classifier [1] since it consists of several stages and evaluates a patch sequentially with a set features in the stages. If a patch is failed in a stage, the remaining stages are not processed. If a patch passed all the stages in the cascade classifier, it is classified as a face (Figure 4(c)).

After scanning is finished, multiple rectangles around the same face, indicating the successful detection, are produced since various sizes of the same image had been scanned. These rectangles are called *candidates* and they are grouped to remove the redundant rectangles.

#### 3.2.2 Face Recognition

LBP based face recognition is based on the LBPH algorithm [2]. The LBPH algorithm consist of mainly three parts as shown in Figure 5: LBP, Histogram, Chi-Distance. Among these parts, Chi-Distance takes 99% of the execution time. First, LBP operation is performed on every training images. Then, the histogram of the LBP is obtained on each image. In the histogramming process, the input image is partitioned into a grid of regions and each region in the grid is histogrammed separately. After processing for all regions, the histograms are concatenated to form a single feature.

![Figure 4. (a) Resizing, (b) Scanning, and (c) cascade classifier in Scanning](image4.png)

![Figure 5. The pipeline of LBP-based face recognition algorithm](image5.png)
This is called \textit{spatially enhanced histogram}, which is the output of training phase of the LBP based face recognition. In prediction step, the testing image is transformed to an LBP feature similarly as the training images were. Then, the similarity of the testing feature is compared with each of training features by Chi-square distance. Chi-square distance is defined as follows.

\[ \chi^2(x, \xi) = \sum_{ij} w_j \frac{(x_{ij} - \xi_{ij})^2}{x_{ij} + \xi_{ij}} \]  

(1)

where \( x \) and \( \xi \) are feature histograms to be compared and indices \( i \) and \( j \) refer to \( i \)-th bin in the histogram corresponding to the \( j \)-th region, and \( w_j \) is the weight for the region \( j \). The person with the lowest distance in the training image is regarded as the same person in the testing image.

LBP based face recognition is feature based approach as opposed to holistic approach where a collection of the training images as a whole produces a set of features. If a new training image is added, the whole images need to be trained again. However, testing requires only constant processing time. In contrast, LBP produces one feature per training image and only the new image is trained and added.

The integrated system processes the detection first, and then performs the recognition on the detected faces, where both use the same LBP feature. In a detection module, a set of scaled images which include various sizes of faces are transformed to LBP features. And the recognition module requires the LBP feature of detected faces. Therefore, the same feature does not need to be calculated twice but the recognition module just reuse the pre-calculated LBP feature from the detection module. However, in order to reuse LBP feature between two modules, it would be needed to resize the detected faces into the same size as the training faces since the size of detected faces varies, and such resizing would incur overheads to the processing time.

Meanwhile, after the detection has finished, it already has the several resized images and each resized image contains resized detected faces. These detected faces have deterministic size that depends on the configuration parameters of the detection process and will not change at runtime. Therefore, we train images after resizing to the agreed specific size in the detection module. Then, we do not need additional resizing after the detection process as the detected face is already in the size of facial image in the training DB.

In a specific application, such as face tracking in a video sequence, it may not need to detect or recognize a face for every frame. However, for the general applications, we assume both face detection and recognition is processed for every frame in an integrated fashion.

4. THE PROPOSED OPTIMIZATIONS

We propose a real-time integrated face detection and face recognition method using LBP feature on embedded GPGPUs. We first parallelize the data-parallel steps explained in the previous section, and applied several optimization techniques to achieve real-time processing on both target GPUs. First, an integrated face detection and recognition is described, and then, the detailed explanation of how the LBP based face detection and recognition are parallelized and optimized are followed. The optimization techniques are applied to both CUDA and OpenCL implementations but we will use OpenCL terminology such as work-item or work-group in the following sections.

4.1 Integrated Face Detection and Recognition

The integrated system processes the detection first, and then performs the recognition on the detected faces, where both use the same LBP feature. In a detection module, a set of scaled images which include various sizes of faces are transformed to LBP features. And the recognition module requires the LBP feature of detected faces. Therefore, the same feature does not need to be calculated twice but the recognition module just reuse the pre-calculated LBP feature from the detection module. However, in order to reuse LBP feature between two modules, it would be needed to resize the detected faces into the same size as the training faces since the size of detected faces varies, and such resizing would incur overheads to the processing time.

Meanwhile, after the detection has finished, it already has the several resized images and each resized image contains resized detected faces. These detected faces have deterministic size that depends on the configuration parameters of the detection process and will not change at runtime. Therefore, we train images after resizing to the agreed specific size in the detection module. Then, we do not need additional resizing after the detection process as the detected face is already in the size of facial image in the training DB.

In a specific application, such as face tracking in a video sequence, it may not need to detect or recognize a face for every frame. However, for the general applications, we assume both face detection and recognition is processed for every frame in an integrated fashion.

4.2 Face Detection on Embedded GPGPUs

4.2.1 Parallelization

We parallelized Grayscale, Resize, LBP and Scanning function into CUDA and OpenCL kernel functions. These are embarrassingly parallel, therefore, implementing the baseline kernels is straightforward: one output pixel per one work-item. In case of Scanning, one work-item evaluates a single patch. The baseline kernels achieve about 5 times speedup on server GPU, however, are severely slower than the single threaded CPU version on the embedded GPUs. This is due to the more limited hardware resources such as smaller number of registers per work-item and the processing elements, lower memory bandwidth, and so forth.

In order to achieve real-time processing on embedded GPUs, we propose the following optimizations.

4.2.2 Skipping with Multi-Phase Scanning

The output of the evaluating a patch in Scanning is a scalar value which corresponds to the number of stages it passed. As shown in Figure 6, the passed number of stages in the cascade tends to increase before the window that detects a face, and decrease after the window.

![Figure 6. Number of passed stages in the cascade classifier as the search window slides](image)

The C/C++ face detector in OpenCV exploits this characteristic, assuming the sequential execution of Scanning on a CPU. If an evaluation on the current patch fails to detect a face at the very first stage of the cascade classifier, then it skips evaluating the next patch since it is very unlikely for the patch to have any face. With this optimization, many patches, or search windows, do not need
to be evaluated, resulting in a large performance gain in execution time.

However, since it may actually have a face in the next patch though it is unlikely, this optimization could incur small error in detection accuracy.

As the optimization is based on the assumption of sequential processing, it is not applicable to the GPU kernel where multiple windows are processed in parallel. In order to apply a similar optimization to GPU, we propose a multi-phase scanning kernel, which consists of multiple kernels for Scanning: phase_0, phase_1, ..., phase_N. Instead of evaluating all the patches in a single kernel, the evaluation is sequentially processed by N kernels, each of which processes only one Nth of the total patches. As shown in Figure 7, the first kernel, phase_0, evaluates (mod N = 0) indexed windows. Each work-item in the kernel evaluates the assigned window and writes the number of passed stages as in the original Scanning kernel. phase_i kernel handles (mod N = i) indexed windows, where 1 ≤ i ≤ N − 1. Each work-item in phase_i except phase_0 evaluates a window only if the previous patch which was already evaluated at phase_(i-1) was not rejected at the first stage.

![Figure 7. Multi-phase kernel launching](image)

Skipping reduces the number of windows to be evaluated but sequential execution of multi-phase kernels can decrease the execution time compared to the simultaneous execution in the original kernel. If the GPU is powerful enough to process all the patches simultaneously in the original kernel, then there would be no gain by skipping through multi-phase kernels, but in fact the execution time would increase due to the sequential execution, as well as the loss in detection accuracy. Depending on the throughput of an embedded GPU and the number of patches in a high resolution image, we can vary the number of phase, N, to figure out the optimal improvement in the trade-off.

4.2.3 Aggregated Kernel Launching

An input image is resized into multiple smaller images as shown in Figure 4(a). For example, a HD input is resized into 17 images with scale factor of 1.2. As explained in Section 3.1, Resizing is followed by LBP and Scanning. Thus, Resizing, LBP and Scanning kernels are called 17 times per each HD input image. Embedded GPUs including Mali have a considerable overhead for kernel launching compared to server GPUs. The overhead can be amortized if all 17 images can be processed in a single kernel: for LBP, all the resized images are put in a single buffer and passed to the kernel at once. More work-items are populated, fully utilizing the GPU. This optimization is applied to Resize, LBP, and Scanning kernel, by which the factor loop in Figure 2 is removed.

4.2.4 Reducing Memory Loads

In LBP kernel, each work-item is assigned one center pixel to produce a binary pattern and examines the value of the neighboring pixels. Since the neighboring pixels are overlapped for the adjacent work-items, it is desirable to remove the redundant memory loads to improve the performance. To do so, we can leverage the shared memory in Tegra K1 GPU so that all CUDA threads first loads their input pixel to the shared memory, then reuse them in generating the binary pattern. However, Mali GPU does not have local memory (i.e. shared memory in CUDA term) and cannot share input pixels with the other work-items in the same work-group. Thus, we map multiple center pixels to a single work-item so that the overlapped loads in the original mapping can be removed. For example, if we map 2x2 center pixels into one work-item as shown in Figure 8, then only 16 memory loads are required. In contrast, if one center pixel is mapped to one work-item, then 36 (=4x9) memory loads would be required with redundant loads since private memory, or registers, cannot be shared with other work-items. As a result, 2x2 mapping would result in 56% reduction in memory accesses.

![Figure 8. 2x2 center pixels for one work-item](image)
more efforts were made to fully utilize the bandwidth by packing small data into larger one and by using OpenCL vectors.

If the kernel loads or stores data in the consecutive global memory space and there exists a larger data type, then, the multiple elements in a small data type can be implicitly cast, or packed, to the larger one without any overhead within the kernel. This enables higher utilization of the bus bandwidth. For example, an array of short can be implicitly cast to an array of integer, which halves the number of memory operations. The packed data can be easily restored by performing a logical and operation followed by a shift operation. This optimization is applied to Resize, LBP, Scanning kernel.

Mali GPU has 128-bit wide vector engine per ALU and it can be easily utilized by using OpenCL vector types and operations. It can not only load and store multiple elements, but also perform arithmetic operations at once. This optimization is applied to Grayscale, Resize, and LBP kernel. In contrast, Tegra K1 GPU does not have a vector engine, nor does CUDA support vector operations in the language. Thus, only multiple loads and stores are reduced by using packing in Tegra K1.

4.2.6 Reducing Data Transfer Overhead

As mentioned in Section 3.1, embedded GPUs can share a memory buffer with the CPU, instead of copying one from another. In OpenCL, clEnqueueMapBuffer() maps the address of the allocated OpenCL device buffer onto a CPU accessible memory address. As a result, both CPU and GPU can read and write on the same memory space. Note that CUDA 6.0 also provides Unified Memory with cudaMallocManaged(), aiming mainly at increasing the productivity rather than the efficiency, by not having separate buffers in the CPU and the GPU. However, in Tegra K1 it can actually improve also the efficiency since it has physically unified memory.

4.3 Face Recognition on Embedded GPGPUs

As described in Section 4.1, the resizing and LBP operation are omitted from the recognition process in the proposed integrated system, which leaves us only Histogram and Chi-distance.  

4.3.1 Histogram

Histogram is very difficult to accelerate on GPUs, even on the server GPUs, since the arithmetic intensity is very low as it has two loads, one addition, and one store. Moreover, synchronization is required to prevent a data race in updating the histogram: Multiple work-items can increase the value of the same bin at the same time. The synchronization can implemented using an atomic operation, or using shared memory and barrier synchronization primitives to perform parallel binary tree reduction.

When the size on which the detection process and the recognition process agreed is large, the processing time of histogramming is not negligible for the real-time processing. Thus, we choose the smallest size among the ones that does not decrease the recognition accuracy.

4.3.2 Chi-square Distance

Parallelizing Chi-Distance requires a sum reduction for the sigmas in Equation (1), which accumulate all the partial Chi-square distances between two bins in each histogram for each of the regions in the test and training image. The partial result computed with the two bins is kept in a floating-point type. However, OpenCL 1.1 does not support atomic operations for the floating-point type. Also as described in Section 3.1, Mali GPU does not provide fast local memory that can be used for parallel binary tree reduction. For this reason, we do not fully implement Chi-Distance on Mali GPU, but perform the sum reduction on the CPU.

In contrast, CUDA supports atomic operations for floating-point types and the sum reduction can be implemented in the kernel with the atomic operation or using shared memory for parallel binary reduction. Three different mappings are used: Mapping each training image to a thread, mapping each histogram to a thread, and mapping each bin in a histogram to a thread. The first version would be helpful when the training DB is huge. No synchronization is required among threads since each thread produces its own distance in this mapping.

In the second version, a thread computes the partial distance of two histograms, each of which belongs to the test image and the training image respectively. For example, when 4x4 grid is used for an image, 16 threads compute each of the partial distances first, and then, the sum reduction is performed using atomic addition to calculate the final distance between two images.

In the third version, a thread computes a partial result with the two bins, one from the histogram in the test image and the other from the training image. Thus, 256 threads compute a partial distance for a single histogram. For 4x4 grids per image, the final distance between two images are obtained from 16 histograms, hence 16 thread blocks whose block size is 256 produce the final distance. This version provides the highest degree of parallelism but requires largest number synchronization in two levels of sum reductions: among the 256 threads in a thread block for a histogram, and among the thread blocks that are dedicated to the same image.

Similar to this various mappings in CUDA for Tegra K1 GPU, we can vary the number of bins to be processed per work-item in OpenCL for Mali GPU. Larger number of bins per work-item can alleviate the sum reduction overheads in CPU since multiple bins assigned to a work-item can be locally accumulated without using any synchronization primitives. However, this reduces the degree of the parallelism and could increase the overall execution time.

5. EXPERIMENTS

Table 1 summarizes the specification of the target platforms. We used HUINS Achiro5250 for OpenCL and Jetson TK1 for CUDA. The kernel device driver version and the user space driver version for Mali T604 are the same as
r3p-02rel0, and arm-none-linux-gnueabi-g++ 4.6.3 was used. For Tegra K1, Linux4Tegra R19.2 and OpenCV4Tegra 2.4.8.2 were used, and arm-linux-gnueabihf-g++ 4.8.2 was used for the CPU and nvcc 6.0.1 was used for the GPU. Since both CPU and GPU run based on DVFS, we set the highest frequency for CPUs and GPUs: The performance governor is used in CPUs.

Table 1. Specification of the Target Platforms

<table>
<thead>
<tr>
<th>Spec</th>
<th>OpenCL Board</th>
<th>CUDA Board</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Cortex-A15 1.7GHz Dual-core</td>
<td>Cortex-A15 2.32GHz Quad-core</td>
</tr>
<tr>
<td>GPU</td>
<td>Mali T604 533MHz GK20a 852MHz</td>
<td></td>
</tr>
<tr>
<td>System Memory</td>
<td>2GB 800 MHz</td>
<td>2GB 933MHz</td>
</tr>
</tbody>
</table>

5.1 Face Detection

As an input for face detection and recognition, a movie trailer which consists of 3000 frames was used. The detection accuracy in a GPU implementation is defined as the number of the total detected faces in the whole 3000 frames divided by the number of the total detected faces of the CPU implementation which is 2189, when no skipping optimization is applied for the same 3000 frames. Note that the ground truth with regard to the number of faces in the input frames is around 2100.

Table 2. Parameters of Face Detection

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scale factor</td>
<td>1.2</td>
<td>xstep</td>
<td>factor&gt;2.0?1:2</td>
</tr>
<tr>
<td>ystep</td>
<td>factor&gt;2.0?1:2</td>
<td>minSize</td>
<td>30x30</td>
</tr>
<tr>
<td>Input Size</td>
<td>HD(1280x720)</td>
<td>maxSize</td>
<td>720x720</td>
</tr>
</tbody>
</table>

The parameters we used for face detection are shown in Table 2. They affect the amount of computation, detection speed and accuracy significantly. The name of each parameter came from the OpenCV face detection. The factor loop that includes Resizing, LBP and Scanning as shown in Figure 2 maintains a loop variable named factor, which has an initial value of 1.0 and represents the ratio of the original input size to the resized input size in the current loop iteration. For each iteration, it is multiplied by a scale factor parameter, resulting in smaller image size, or larger patch size in the other perspective. The loop ends when the scaled patch size becomes larger than maxSize. Note that the patch is not actually scaled but fixed as explained in Section 3.2.1. If the scaled patch size is less than minSize, the factor loop is skipped. For the VGA input image, the loop iterates 15 times with the scale factor of 1.2, and 17 times for HD input image.

The ystep and xstep parameters are also used in Scanning to determine how densely the search should be made. They indicate the stride between two consecutive windows.

In Scanning kernel, which is the most time-consuming one, the skipping with multi-phase kernels for VGA inputs was applied varying N, and the results as well as the CPU skipping result are shown in Figure 9. CPU1 and CPU2 denote the CPU versions without and with skipping respectively. Since CPU version is inherently sequential, the execution time is reduced by 47% with 2% accuracy loss.

Figure 9. Trade-off between execution time and accuracy in skipping with multi-phase Scanning kernel on Mali GPU (VGA)

In the multi-phase kernel with N=2, the execution time is reduced by 20% compared to N=1 version, and the detection accuracy is reduced by 2%. As N increases, the execution time does not decrease further but starts to increase. This is because the amount computation is not heavy compared to the throughput of Mali GPU. Tegra K1 has higher throughput and the execution time does not decrease either.

Figure 10. Aggressive skipping with multi-phase Scanning kernel on Mali GPU (VGA)

Figure 10 shows the results when more aggressive skipping with multi-phase kernels is applied. In this method, if phase_i finds out the patch to be far from a face, then the remaining kernels (phase_(i+1), ..., phase_N) do not evaluate their patch at all. This method of course loses the detection accuracy but as the trade-off the execution time decreases further. With N=5, the execution time is reduced by 40% and the detection accuracy is reduced by 10%, in spite of which, there was no significant defects compared to N=1 version when the video is played with the rectangles of detected faces.

Figure 11. Trade-off between execution time and accuracy in skipping with multi-phase Scanning kernel on Mali GPU (HD)

Figure 11 shows the results with HD inputs. Similarly, execution time is reduced when N=2 by 10% but still increases again as N gets larger than 2. In LBP kernel, 8 pixels are required to compute one output pixel for the
default mapping of one center pixel to one work-item. When we mapped multiple pixels to a work-item, the execution time of LBP decreased as the memory loads of the overlapped pixels are avoided. Figure 12 shows the speedup of LBP kernel compared to the CPU implementation when varying the number of center pixels per work-item. The speedup time increases until the number of pixels are 2x2, meaning that the gain from the memory loads reduction is larger than the loss from the reduced parallelism. We could not increase the number of center pixels to be mapped further due to the constraint on the number of registers that a work-group can use, which is much tighter on embedded GPUs than on server GPUs. As shown in the figure, when 4x4 pixels are mapped, the speedup drops due to the reduced number of simultaneous threads running on the GPU. Note that this optimization does not help when a kernel processes only one image: It helps only after the memory loads become a bottleneck as in the kernel for the aggregated images. With additional improvement using OpenCL vectors, the final LBP kernel achieves 6.6 times speedup on Mali GPU.

For Tegra K1 GPU where faster shared memory is available, it turned out to be faster to use the shared memory without sacrificing the parallelism. 2x2 center pixels per thread implementation takes 1.78 msec whereas 1 pixel per thread implementation with shared memory takes 0.57 msec, resulting in more than 3 times speedup.

![Speedup of LBP kernel on Mali GPU](image)

**Figure 12.** The speedup of LBP kernel on Mali GPU when increasing the number of center pixels to a work-item

In Resizing kernel, four pixels are interpolated to produce a single output pixel. Thus, it needs to load four pixel values and the four interpolation weights. This is highly memory-intensive, hence we apply packing and vectorization. It achieved 2.67 times speedup.

Since all the three major bottlenecks have been successfully accelerated, we further parallelized and optimized Grayscale. In Grayscale kernel, it requires 4 memory operations and 2 arithmetic operation per one output pixel. OpenCL vector optimization was applied similarly to Resizing kernel. It achieved the speedup of 1.18.

The speedups of the OpenCL kernels with the proposed optimizations on Mali and Tegra K1 are shown in Table 3 and Table 4 respectively.

Figure 13 summarizes the performance improvement in execution time on Mali GPU as the proposed optimizations are incrementally applied in face detection. As already mentioned, unlike on server GPUs, the baseline OpenCL implementation is even slower than the single threaded CPU version for VGA inputs. When the aggregate kernel launching, skipping with N=2 multi-phase kernel, the memory loads, packing and vectorization, and big table are applied, it results in 2.4 times speedup for VGA inputs, and 2.8 times speedup for HD inputs as shown in Table 3 and Figure 14.

<table>
<thead>
<tr>
<th>Name</th>
<th>CPU (ms)</th>
<th>OpenCL (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grayscale</td>
<td>4.06</td>
<td>3.44</td>
<td>1.18x</td>
</tr>
<tr>
<td>Resize</td>
<td>17.90</td>
<td>6.70</td>
<td>2.67x</td>
</tr>
<tr>
<td>LBP</td>
<td>32.83</td>
<td>4.94</td>
<td>6.64x</td>
</tr>
<tr>
<td>Scanning</td>
<td>59.93</td>
<td>21.14</td>
<td>2.83x</td>
</tr>
<tr>
<td>FPS</td>
<td>8.25</td>
<td>23.50</td>
<td>2.84x</td>
</tr>
</tbody>
</table>

**Table 3.** Execution time and speedup of the proposed face detection kernels on Mali GPU (HD 720p)

<table>
<thead>
<tr>
<th>Name</th>
<th>CPU (ms)</th>
<th>CUDA (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grayscale</td>
<td>2.95</td>
<td>1.78</td>
<td>1.66x</td>
</tr>
<tr>
<td>Resize</td>
<td>11.05</td>
<td>2.78</td>
<td>3.97x</td>
</tr>
<tr>
<td>LBP</td>
<td>26.01</td>
<td>0.57</td>
<td>45.63x</td>
</tr>
<tr>
<td>Scanning</td>
<td>48.52</td>
<td>13.98</td>
<td>3.47x</td>
</tr>
<tr>
<td>FPS</td>
<td>10.81</td>
<td>40.98</td>
<td>3.79x</td>
</tr>
</tbody>
</table>

**Table 4.** Execution time and speedup of the proposed face detection kernels on Tegra K1 GPU (HD 720p)

Figure 14 shows the results of the proposed face detector in each platform including OpenCV’s. The frame per second of the OpenCV CUDA face detector was measured in Jetson board as well as OpenCV CPU face detector. Our detector on Tegra K1 GPU is 4.3 times faster than the OpenCV CUDA detector. Note that there is no OpenCL face detector in OpenCV, except Haar based one. Thus, we could
not compare the performance on Mali GPU. The detection accuracy is maintained as the CPU version of OpenCV face detection.

Figure 15. Comparison of detection speed between OpenCV’s face detection and ours on Tegra K1 GPU and on Mali (fps)

5.2 Face Recognition

The inputs we used for face recognition are from FERET Database[14] which has 250 images of upper body of 86 people. We manually extracted the faces in each image.

First of all, we examined the possible performance gain with Unified Memory on Tegra K1 GPU, which actually has unified on-chip memory shared with the CPU. For 4MB of train DB transfer, the conventional memory copy takes 13.5 msec while Unified Memory takes 11.8 msec. With the increased DB of 16MB size, the conventional copy takes 50 msec while Unified Memory takes 16 msec, which confirms the reduced data transfer on embedded GPUs.

Then, only Histogram and Chi-Distance results are shown. Resizing and LBP can be omitted as mentioned in Section 4.1 since the training images have been already trained for the agreed size with the detection module: The size is one of the detected face sizes. We varied the size in order to check the recognition accuracy drop in Figure 16.

The recognition accuracy was measured using leave-one-out cross-validation method. With 250 training images, we take one out from the training images and designate it as a test image. Thus, there are 250 predictions and the accuracy is calculated as the number of the correct prediction over 250. Note that the size shown in Figure 16 is the possible face sizes among the face detection outputs.

Figure 16. Recognition accuracy when varying the facial image size

The results show that the resizing does not severely affect the accuracy until some point. The detection accuracy is severely dropped at the size of 461 because the number of noise pixels generated by resizing overwhelmed the number of the original pixels which contain crucial facial information. Therefore, there is no reason to choose large face size which would only increase the processing time for Histogram. We chose the face size to be 68x68, and Histogram with a 68x68 image takes no more than one msec even on ARM Cortex-A15 CPU with 1.7 GHz.

On the other hand, The Chi-Distance kernel is the most time-consuming one with high arithmetic intensity; Two subtraction, one addition, one multiplication, and one division which adds to 5 ALU operations whereas only two loads are needed. However, the baseline OpenCL implementation, denoted as casting, was slower than the CPU version, as shown in Figure 17. It was due to the type castings in the kernel. In order to calculate the Chi-square distance which involves a division, the data type of each bin must be cast to a floating-point type from an integer type. After removing such type casting inside the kernel by processing the histograms with a floating-point type, instead of an integer type in the first place, it achieved 1.7 times speedup. Note that such degradation due to implicit casting inside the kernel was not observed in CUDA 6.0 for Tegra K1 GPU.

Figure 17. Execution time of Chi-Dist kernel on Mali GPU when a work-item processes multiple bins

As mentioned, OpenCL 1.1 (as well as 1.2) does not support atomic operation for a floating-point type and the Mali GPU does not have faster local memory than global memory. Thus, sum reduction is performed on the CPU after the partial reduction by each work-item. We varied the number of bins to be processed per work-item and the execution time was reduced as number of bins computed by a single work-item increases. This is because Mali GPU was over-utilized by a massive number of work-items. By computing multiple bins in a single work-item, the sum reduction time is reduced without any trade-off. When the
number of bins is 16, the Chi-Distance OpenCL kernel achieves 4.9 times speedup compared to single threaded CPU version.

Figure 18 shows the execution time of the CUDA implementations on Tegra K1 GPU with different mappings described in the previous section. The first version is faster than the second version. This is because only 16 histograms, hence 16 threads runs per thread block. To increase the block density, 16 images are processed, hence 256 threads run per block, resulting in a similar speedup as the first version. The third version is the fastest, meaning that providing more parallelism than reducing the number of sum reductions is advantageous to the speedup. The sum reduction by parallel binary tree reduction on shared memory is faster than the one with atomic operations either on global or shared memory.

Finally, the end-to-end execution time of the proposed integrated face detection and recognition on Tegra K1 GPU is measured, and it is 38 fps for HD inputs and 99 fps for VGA inputs, which is 3.7 times and 2.8 times speedup compared to the single threaded CPU version. The results on Tegra K1 GPU are shown in Table 5. On Mali GPU, it is 21.9 fps for HD inputs and 55.6 fps for VGA inputs, which is 2.9 times and 2.7 times speedup as shown in Table 6.

Table 5. Execution time and speedup of the proposed face recognition kernels on Mali GPU (HD 720p)

<table>
<thead>
<tr>
<th>Name</th>
<th>CPU</th>
<th>OpenCL</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Histogram</td>
<td>0.05</td>
<td>0.5</td>
<td>0.1x</td>
</tr>
<tr>
<td>Chi-Dist.</td>
<td>12.61</td>
<td>2.59</td>
<td>4.86x</td>
</tr>
<tr>
<td>FPS</td>
<td>7.57</td>
<td>21.85</td>
<td>2.89x</td>
</tr>
</tbody>
</table>

Table 6. Execution time and speedup of the proposed face recognition kernels on Tegra K1 GPU (HD 720p)

<table>
<thead>
<tr>
<th>Name</th>
<th>CPU</th>
<th>CUDA</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Histogram</td>
<td>0.15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chi-Dist.</td>
<td>6.57</td>
<td>1.67</td>
<td>x3.93</td>
</tr>
<tr>
<td>FPS</td>
<td>10.47</td>
<td>38.47</td>
<td>x3.67</td>
</tr>
</tbody>
</table>

6. CONCLUSION

Face detection and recognition is one of the most popular applications these days as CCTVs proliferate and smart glasses will become prevalent soon. Many researchers have been conducted to accelerate the efficiency of the face detection but not on embedded GPUs. In this paper, we have proposed several optimizations that can fully utilize the embedded GPUs to achieve real-time processing of the integrated face detection and recognition for HD images: skipping with multi-phase kernels in Scanning, careful mappings for reducing loads in LBP, etc. In addition, different mappings and their trade-off in terms of the degree of parallelism and the reduction cost is examined to find out the optimal implementations.

As a result, the integrated face detection and recognition achieves 21.9 fps on Mali GPU and 38.47 fps on Tegra K1 GPU, which are 2.9 times and 3.7 times faster processing than the CPU versions respectively.

7. ACKNOWLEDGEMENT

This work was supported by the IT R&D program of MSIP/KEIT (10044494).

8. REFERENCE