Using Mobile GPU for General-Purpose Computing – A Case Study of Face Recognition on Smartphones

Kwang-Ting (Tim) Cheng, Yi-Chu Wang
University of California, Santa Barbara, CA, U.S.A
timcheng@ece.ucsb.edu, yichuwang@umail.ucsb.edu

ABSTRACT

As GPU becomes an integrated component in handheld devices like smartphones, we have been investigating the opportunities and limitations of utilizing the ultra-low-power GPU in a mobile platform as a general-purpose accelerator, similar to its role in desktop and server platforms. The special focus of our investigation has been on mobile GPU's role for energy-optimized real-time applications running on battery-powered handheld devices. In this work, we use face recognition as an application driver for our study. Our implementations on a smartphone reveals that, utilizing the mobile GPU as a co-processor can achieve significant speedup in performance as well as substantial reduction in total energy consumption, in comparison with a mobile-CPU-only implementation on the same platform.

INTRODUCTION

Exploring the use of Graphics Processor Unit (GPU) as a general-purpose co-processor to accelerate compute-intensive applications has been an active research subject in the past few years. Depending on the applications, the underlying algorithms' inherent parallelism, and the computing capability offered by the GPUs, a speedup ranging from several times to hundreds of times has been reported in the literatures. The programmable GPU is now also available in handheld devices, such as smartphones and portable gaming devices. Since those handheld devices are powered by batteries, the mobile GPU are typically designed with limited power ceiling of less than 1 watt [6]. As a result, the mobile GPU usually has fewer cores, lower memory bandwidth, and variant architecture when compared to the desktop GPUs.

As most existing CPU-GPU optimizations are based on, and optimized for, desktop and server platforms, it is desirable to characterize the computing capability of a mobile CPU-GPU platform and revisit the GPGPU strategies in order to better utilize the computational power of a mobile GPU. Moreover, it is also important to characterize the power efficiency of the mobile GPU and CPU in order to optimize the total energy consumption when developing a mobile application. There were few papers discussing the utilization of a mobile GPU as a general-purpose co-processor. In [5], the authors implemented an image filtering with a 3x3 kernel on the PowerVR GPU in a mobile SoC, TI's OMAP3 [1], and optimized the performance (from 13fps to 29fps) by reducing the precision of the floating point variables whenever possible. In [4], the authors reported that the PowerVR GPU, operating at 111 MHz, achieved a better performance (0.41fps vs. 0.11 fps) and lower power consumption (232 mW vs. 361 mW) for low-level image processing tasks (such as Gaussian blur, color and geometry adjustments), in comparison with the ARM Cortex-A8 processor running at 555 MHz on the same platform. Their investigation was conducted with an emulated version of OpenCL [9] embedded profile, which, however, is not yet available for current commodity smartphones and development boards.

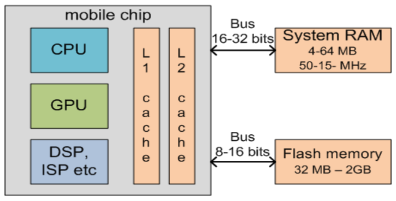
In this paper, we investigate the computational capability and energy efficiency of a current mobile CPU-GPU system for an exemplar computer vision application - face recognition. We first give an overview of the mobile GPU's architecture and the programming API in Section 2. A comparison of a mobile GPU's performance and its power efficiency to a mobile CPU and a desktop GPU is made in Section 3. Then in Section 4, we describe our GPU-accelerated face recognition application and demonstrate that utilizing the mobile GPU can achieve both significant performance boost and substantial energy reduction for our target application. Finally we conclude our study and discuss future research directions.

MOBILE GPU OVERVIEW

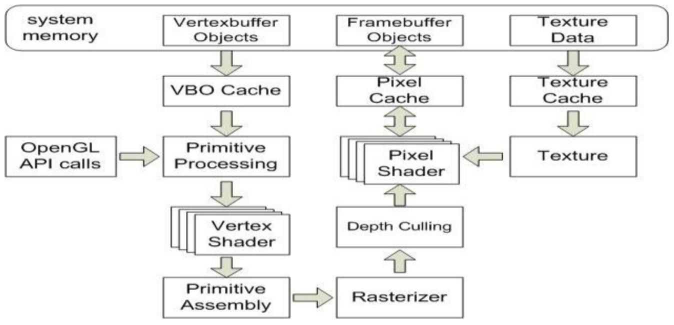
A GPU inside a mobile device is typically integrated into the application processor system-on-a-chip (SoC) which also consists of one or several CPUs, DSP, and other application-specific accelerators, as shown in Figure 1.(a). Instead of having its own graphics memory, an embedded GPU shares the system bus with other computing cores to access the external memory and therefore has much lower memory bandwidth than those of laptop and high-performance desktop systems [6]. The major SoCs and the mobile GPUs available in the market include Qualcomm's SnapDragon SoC with the Adreno 200 GPU [2], TI's OMAP3 SoC with the PowerVR SGX 530/535 [1], and Nvidia's Tegra2 SoC with its own ultra-low-power (ULP) version of GeForce GPU [3].

Mobile GPUs are usually designed with emphasis of lower power consumption rather than performance. Reducing the traffic between the GPU and the memory is one of the key techniques to reduce the power consumption in the architecture level design [6]. For example, the GeForce GPU inside Tegra, as shown in Figure 1(b), has implemented on-chip caches for pixel, texture, vertex, and attribute to reduce the system memory transactions during the rendering process. The inclusion of caches provides better performance since cache access has less latency than off-chip memory [3]. Storing the entire texture and frame buffer, which are used in the rendering pipeline, in the on-chip buffer could significantly reduce memory transactions. To make it feasible, the PowerVR GPU exploits a *tiling* architecture which divides a frame into small tiles (e.g. a rectangular block, usually 16x16 pixels) and renders one tile at a time so that all the required data can be stored in the on-chip memory [6].

Another way to reduce the number of memory transactions is to store the compressed data (e.g. compressed textures, vertex and frame buffers) in memory and then decompressed on-the-fly by GPU cores before processing [7]. Avoiding unnecessary memory access also helps to lower the power consumption. In 3D graphics rendering, multiple polygons and their corresponding pixels may overlay the same 2D screen-based pixel locations. Traditional graphics pipeline usually renders all the polygons, including the occluded ones, and then "*culling*" adequate polygons to display according to the polygons' depths. The Tegra GPU, as illustrated in Figure 1.(b), moves the culling stage before rendering all the objects so as to effectively minimize unnecessary computations as well as unnecessary memory access.



(a)



(b)

Figure 1. (a) A high-level block diagram of a mobile SoC system. The GPU shares the system bus with the CPU and other computing hardwares. (b) The processing flow and block diagram of an ultra-low-power GeForce GPU in the Tegra SOC [3]. The inclusion of caches and performing the depth culling before the pixel processing help reduce the traffic to system memory and reduce the overall power dissipation.

Programming API

OpenGL ES 2.0 [8] is the primary graphics programming interface for handheld and embedded devices with a programmable GPU: a programmable vertex shader and fragment shader for per vertex position and per pixel calculation respectively. Other stages in the rendering pipeline, such as rasterization and culling, remain as fixed functions. OpenGL ES 2.0 is a subset of the widespread adopted OpenGL standard used in desktop systems and game consoles. This subset removes some redundancy from the OpenGL API. For example, if multiple methods can perform the same operation, the most useful method is adopted and other redundant methods are removed. An example is that only vertex arrays are used to specify geometry in OpenGL ES 2.0, whereas in OpenGL an application can also use the immediate mode and the display lists in addition to the vertex array to specify geometry. There are also new features introduced to address specific constraints of handheld devices. For example, to reduce the power consumption and increase the performance of the shaders, precision qualifiers were introduced to the shading language: *lowp* (10-bit fixed point format in the range [-2,2), with a precision of 1/256), *mediump* (16-bit floating point values in the range [-65520, 65520]), and *highp* (32-bit floating point variables) [11].

To utilize a mobile GPU as a general-purpose accelerator, programmers have to map the algorithms to the graphics operations, and write the shader programs to configure the vertex and fragment shaders. However, the graphics APIs expose little controllability to the low-level hardware, and hence makes it less flexible to use the GPU for general-purpose computing. For example, the graphics APIs in the current versions of OpenGL ES do not have the "scatter" operation (i.e. write to an arbitrary memory location), or thread-level synchronization. The commonly used high-level APIs for a desktop environment, such as CUDA [10] and OpenCL [9], are not supported in the embedded platform yet.

MOBILE GP-GPU: CAPABILITY AND LIMITATIONS

We implemented Fast Fourier Transform (FFT), a kernel computation of many image and signal processing algorithms, on a mobile GPU with the intent of testing the applicability of utilizing mobile GPU for higher level tasks. Its performance and power consumption were then compared to a mobile CPU and a desktop GPU. Our experiments were performed on an Nvidia Tegra SoC [3] with the following specifications: a 1GHz dual-core ARM Cortex-A9 CPU, 1GB of RAM, an Nvidia ultra-low-power GeForce GPU running at 333MHz, and 512MB of Flash memory.

FFT on GPU

A 1D FFT of sample N is defined as $X(k) = \sum_{n=0}^{N-1} x(n) e^{-2\pi nk/N}$, where $0 \le k \le N-1$. In our study, we examined an implementation of the Cooley-Tukey FFT algorithm, based on the approach presented in [15], using OpenGL ES 2.0 API and the shader language. The processing flow of the Cooley-Tukey method is depicted in Figure 2. A total of $log_2 N$ stages are required to complete the computation for N samples. Samples in each stage form pairs between two groups, as shown in the dotted box of Figure 2. The computation of this group is expressed as: $c = a + w^0 * b$ and $c = a - w^0 * b$. The coefficients of each sample (i.e. $\pm w^0$) is pre-computed and stored as a texture for the shader program to fetch. It should be noticed that the sign of the coefficient is also included in the texture in order to avoid the conditional computation (e.g. branches) within the shader program.

The transformation of a 2D image is done by applying 1D FFT to rows and columns consecutively. For each stage, a quad is rendered covering the entire 2D $N \times N$ data array. Each stage requires a texture of size $N \times 1$ to store the pre-computed coefficients. To reduce the memory bandwidth between system memory and GPU memory, the coefficients for the real part and the imaginary part are stored in the two channels of the texture. Another texture storing the fetch indices is required. Although it could be combined with coefficient texture, such implementation is wasteful because indices require a lower resolution texture than that required for coefficients. Therefore, we allocate a texture with a lower resolution for fetch indices to reduce the memory bandwidth. The iterative processing is implemented by multiple rendering passes, with a floating point framebuffer object (FBO) (Chapter 12 in [11]) to store the intermediate rendering results.

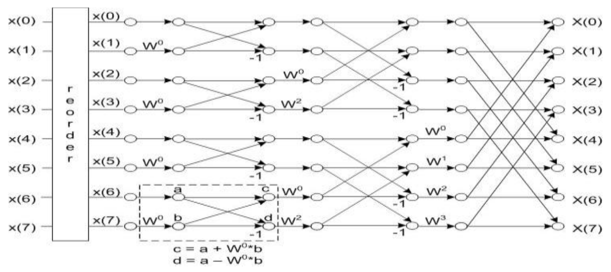


Figure 2. Processing flow of a 8-sample 1D FFT

Comparison With Mobile CPU

The measured execution time and measured power consumption of computing 2D complex FFT of various sizes on the Tegra CPU and GPU are shown in the first and second rows respectively of Table 1. The listed execution time is the average time of computing FFT and IFFT 50 times. For 128×128 FFT, the GPU is 3x faster and consumes 8% more power than the CPU (1 second vs. 3.1 seconds, and 4.1 watts vs. 3.7 watts). The slightly higher power when using GPU is because the CPU is not idle when the GPU is running and is standing by for the completion of GPU. As a result, the ratio of the total energy consumption of the CPU version vs. the GPU version for this FFT benchmark is 2.86 to 1. The speedup of GPU over CPU decreases to 2.2X and 1.3X when computing 256×256 and 1024×1024 FFT respectively. For a large-size FFT, data cannot be fit into the cache and more off-chip memory transactions would be required. Since CPU usually has a larger cache than GPU, it results in better performance for a larger size FFT. Moreover, while GPU core is waiting for data from memory to compute, it is not running at the full utilization. Therefore, the power consumption of running a larger size FFT is slightly lower than that of running a smaller size FFT (3.6 watts vs. 4.1 watts.)

Comparison With Desktop GPU

Next, we compared the performance and power efficiency of the ULP GeForce GPU with Nvidia's GeForce 8800 GPU, showing in the second and third rows respectively of Table 1. For both mobile GPU and desktop GPU, the listed power consumption is the measured power of the whole system. The desktop GPU power dissipation listed in the table is the measurement results reported in [16]. The FFT execution time on GeForce 8800 was reported in [17].

Although spending much longer time to complete the FFT task than the desktop GeForce 8800 GPU, the ULP GeForce GPU consumes significantly less power too. When performing a smaller-sized FFT (e.g. 128×128), the ULP GeForce GPU can actually achieve better energy efficiency (41 mJ vs. 53 mJ). For larger-sized FFT, this energy advantage disappears due to the limited cache size for a mobile GPU.

		128*128	**256*256**	**1024*1024**
Tegra CPU (ARM Cortex A9)	Time (ms)	31	124	1953
	Power (W)	3.67	3.69	3.57
	Energy (mJ)	113.7	457.5	6,972.2
Tegra GPU (ULP GeForce)	Time (ms)	10	55	1450
	Power (W)	4.10	3.70	3.65
	Energy (mJ)	41.0	203.5	5,292.5
GeForce 8800 + Intel i7 [16]	Time (ms)	0.11	0.17	1.74
	Power (W)	483.89	483.89	483.89
	Energy (mJ)	53.2	82.11	841.96

Table 1. Execution time and power consumption comparison of the mobile GPU, mobile CPU, and desktop GPU. The power consumption is measured for the whole system: for Tegra, the power consumption numbers are for the Tegra devboard; for a desktop system, the power consumption numbers are the total power consumed by the CPU, motherboard and attached graphics card.

Trade-off and limitations

A GPU relies on the CPU to generate vertex and texture data, to transfer data to the GPU memory, and to launch the GPU execution. For the FFT computation, the initialization of the GPU takes about 0.4 second. To benefit from the potential speedup that a GPU can provide, the task running on a GPU should be compute-intensive to offset the overhead of this initialization time.

Moreover, floating point texture as well as framebuffer objects (FBO) are required for the FFT computation and for many other iterative computations with a floating point precision requirement. In OpenGL ES 2.0, the floating point textures are only supported if the implementation exports *OES_texture_float* extension. However, the support of *OES_texture_float* does not necessary mean it supports floating point FBOt. While the floating point buffer can be allocated, the driver may not expose it to the application developers. Among the three major mobile GPU cores, Nvidia Tegra [3] , PowerVR SGX [1], and Adreno [2], only Tegra supports floating point FBO. Without the floating point FBO, developers have to either use fixed point computation and sacrifice the accuracy, or convert a floating point value to the RGBA8888 representation (so that the value conforms to the format for display) for storage in the buffer, and convert it back to the floating point format after fetched from the buffer. These conversions incur significant time overhead and may completely offset the benefits of utilizing GPU.

APPLICATION: MOBILE FACE RECOGNITION

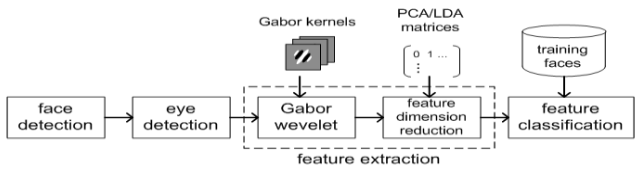
Figure 3(b) shows a smartphone application of face recognition: identifies a known person in a given photo, tags this person's name, and links it to the address book or uploads the information to a social network account such as Facebook. Face recognition, the enabling technique in this application, consists of four steps which are illustrated in Figure 3(a): (1) Face detection, which scans the entire image to identify face regions. (2) Face landmark localization, which identifies the face landmark regions such as the eyes, the nose and the mouth, and then resizes and registers the face region accordingly. (3) Face feature extraction, which represents a face region by its features that are sufficiently invariant or robust to the variation of illumination, pose, facial expression, and occlusion. (4) Face feature classification, which compares the face feature to the training face set and in turn assigns a name of the most similar identity to the query face.

We implemented a face recognition system with a Gabor-based face feature representation, which is considered as one of the best local face representations. The Gabor wavelet representation of an image is the convolution of the image with a family of Gabor kernels as defined in [12]. A typical Gabor-based face descriptor uses 40 different Gabor kernels, which include 5 different scales and 8 different orientations, and extracts the structure corresponding to multiple-frequency (scale) and multiple-orientation in a local area. The Gabor-face is further divided into several non-overlapping local patches, and then processed by Principle Component Analysis (PCA) and Linear Discriminant Analysis (LDA) to reduce its feature dimensionality and to form the final face feature descriptor. The use of the Gabor feature combined with the LDA-based recognition method has been reported to achieve an accuracy of 93.83% on the traditional face recognition dataset FERET [13].

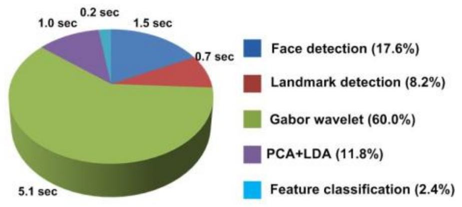
Directly porting the codes designed for the desktop platform to the Tegra platform and running the program on the Tegra CPU takes 8.5 seconds to detect and recognize a person. Figure 3(c) shows the breakdown of the execution time. In this implementation, Android facedetector API [14] is used to identify the face regions in a given photo, and an AdaBoost-based eye localization method is used to identify the landmark regions. The detected face region is resized to 64×84 before extracting face features. Face feature classification is performed using the K-nearest-neighbor method. This long runtime clearly indicates that the face recognition workload is too heavy for a smartphone platform and it is desirable to be accelerated it by at least 5x for offering a satisfactory user experience. Specifically, computing the Gabor representation of a face image is our target of optimization.

GPU Accelerated Face Recognition System

The Gabor wavelet can be implemented by either convolution or the Fast Fourier Transform (FFT). The convolution method is suitable only for small-size kernels due to the memory limitation. However, a small-size kernel is not suitable for object and pattern recognition. Therefore, our GPU-based Gabor face feature extraction is based on the FFT method, which first transforms both face image and the Gabor kernel into the Fourier space, multiplies them together, and then inverse-transforms them back to the space domain.



(a)



(b) (c)

Figure 3. (a) Steps in a face recognition system. (b) Exemplar application snapshot. (c) Execution time breakdown of the face recognition system running on the Tegra platform's CPU.

The experimental results are shown in Table 2. Utilizing the Tegra GPU to compute the Gabor wavelet, the task can be completed in just 1.2 seconds, which represents a 4.25x speedup in comparison with the CPU implementation. With the GPU successfully offload the computational burden from CPU, the overall computation time for recognizing a person on a smartphone is reduced from 8.5 seconds to 4.6 seconds. As for the total energy consumption, the mobile CPU-GPU implementation consumes 16.3J while the CPU only implementation consumes 29.8J. After the Gabor wavelet is accelerated by the GPU, the face detection and face feature dimension reduction by PCA-LDA become the most time-critical parts. We will further explore the opportunity of utilizing mobile GPU to alleviate these new computational bottlenecks to further improve the performance and energy consumption.

Conclusions

In this paper, we investigate the computing power and energy consumption of a mobile CPU-GPU platform for mobile computer vision applications. In our case study, we implemented a GPU-accelerated Gabor face feature extraction, the most compute-intensive task in a face annotation system and achieved a 4.25x speedup. This experimental investigation confirms that a mobile GPU, although designed primarily for low-power rather than maximum performance, can provide significant performance speedup for vision tasks on a mobile platform, similar to the role of its high-performance counterparts in the desktop and server systems. In addition, the performance improvement achieved by GPU-based computing also results in overall energy reduction for the given task, which is a tremendous benefit for mobile devices. Our case study of GPU-assisted face feature extraction demonstrated a 3.98x reduction in total energy consumption, resulting from the facts of a 4.25x speedup and only slight increase in power consumption.

With an ultra-low-power architecture and reduced number of cores, the power consumption of a mobile GPU is typically 2 orders of magnitude lower less than its desktop version (e.g. 4 watts for Nvidia Tegra SoC vs. 483 watts for a desktop system with Nvidia GeForce 8800 GPU). Therefore, although a moible GPU is much less powerful in computation in comparison with a desktop GPU, it might still achieve better overall energy efficiency for completing a given task.

Function	Feature Extraction		Face Recognition	
Method	CPU only	CPU+GPU	CPU only	CPU+GPU
Time (s)	5.1	1.2	8.5	4.6
Energy (J)	18.7	4.9	29.8	16.3

Table 2. Execution time and total energy consumption of the face feature extraction and the entire face recognition application with and without GPU acceleration.

References

[1] Texas Instruments Inc. OMAP3 family of multimedia application processors, http://focus.ti.com.

[2] Qualcomm Inc. http://www.qualcomm.com/snapdragon

[3] Nvidia Corporation, "Bring High-End Graphics to Handheld Devices," Nvidia whitepaper, 2011.

[4] J. Leskela, J. Nikula, and M. Salmela, "OpenCL Embedded Profile Prototype in Mobile Device," In IEEE Workshop on Signal Processing Systems, pp. 279-284, October 2009.

[5] N. Singhal, I. K. Park, and S. Cho, "Implementation and Optimization of Image Processing Algorithms on Handheld GPU," IEEE International Conference on Image Processing, pp. 4481-4484, September 2010.

[6] T. Akenine-Möller and J. Ström, "Graphics Processing Units for Handhelds," Proceedings of the IEEE, vol. 96, Issue 5, pp.779-789, 2008.

[7] J. Ström and T. Akenine-Möller, "iPACKMAN: High-Quality, Low-Complexity Texture Compression for Mobile Phones," in Proc. Graph. Hardware, pp.63-70, 2005.

[8] Khronos Group, OpenGL ES 2.0 Specification, http://www.khronos.org/opengles.

[9] Khronos Group, Open Computing Language (OpenCL) Specification, http://www.khronos.org/opencl.

[10] Nvidia Corporation, "Nvidia Compute Unified Device Architecture (CUDA) Programming Guide," version 2.0, 2008.

[11] A. Munshi, D. Ginsburg, and D. Shreiner, "OpenGL ES 2.0 Programming Guide," Addison-Wesley, USA, 2008.

[12] Y. Su, S. Shan, X. Chen, and W. Gao, "Hierarchical Ensemble of Global and Local Classifiers for Face Recognition," IEEE Transactions on Image Processing, vol.18, no.8, pp.1885-1896, 2009.

[13] P. J. Phillips, H. Moon, S. A. Rizvi, and P. J. Rauss, "The FERET Evaluation Methodology for Race-Recognition Algorithms," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol.22, no.10, pp.1090-1104, 2000.

[14] Open Handset Alliance, Android Software Develop Kit 2.2, http://developer.android.com/intex.html. 2010.

[15] T. Sumanaweera and D. Liu, "Medical image reconstruction with the FFT," In *GPU Gems 2,* Addison-Wesley, pp.765-784, 2005.

[16] D. Ren and R. Suda, "Power Efficient Large Matrices Multiplication by Load Scheduling on Multi-core and GPU Platform with CUDA," International Conference on Computational Science and Engineering, Vancouver, August 2009.

[17] L.D. Brandon, C. Boyd, and N. Govindaraju, "Fast computation of general Fourier Transforms on GPUs," IEEE International Conference on Multimedia and Expo, Hannover, April 2008.