Energy-Aware Real-time Face Recognition System on Mobile CPU-GPU Platform

Yi-Chu Wang, Bryan Donyanavard, Kwang-Ting (Tim) Cheng

Dept. of Electrical and Computer Engineering
University of California, Santa Barbara, CA 93106, USA
yichuwang@umaill.ucsb.edu, bryandony@gmail.com, timcheng@ece.ucsb.edu

Abstract. The Graphics Processor Unit (GPU) has expanded its role from an accelerator for rendering graphics into an efficient parallel processor for general purpose computing. The GPU, an indispensable component in desktop and server-class computers as well as game consoles, has also become an integrated component in handheld devices, such as smartphones. Since the handheld devices are mostly powered by battery, the mobile GPU is usually designed with an emphasis on low-power rather than on performance. In addition, the memory bus architecture of mobile devices is also quite different from those of desktops, servers, and game consoles. In this paper, we try to provide answers to the following two questions: (1) Can a mobile GPU be used as a powerful accelerator in the mobile platform for general purpose computing, similar to its role in the desktop and server platforms? (2) What is the role of a mobile GPU in energy-optimized real-time mobile applications? We use face recognition as an application driver which is a compute-intensive task and is a core process for several mobile applications. The experiments of our investigation were performed on an Nvidia Tegra development board which consists of a dual-core ARM Cortex A9 CPU and a Nvidia mobile GPU integrated in a SoC. The experiment results show that, utilizing the mobile GPU can achieve a 4.25x speedup in performance and 3.98x reduction in energy consumption, in comparison with a CPU-only implementation on the same platform.

1 Introduction

It has been an active research subject to explore the use of Graphics Processor Unit (GPU), an indispensable component in desktop computers, as a general purpose coprocessor to accelerate the compute-intensive part of an algorithm. The research directions include (1) to identify algorithms' parallelism or redesign algorithms to be suitable running on a GPU, and (2) to extend the fixed graphics pipeline into programmable pipelines with a more flexible memory manipulation by high-level APIs, such as CUDA[1]. Depending on the algorithms' inherent parallelism, the number of cores, and the available memory bandwidth of the GPU hardware, a speedup of tens to hundreds has been reported in the literatures for various applications. Computer vision is one of the areas for which

the GPU has demonstrated significant performance improvement, such as image registration [2] and feature tracking [3].

The programmable GPU is now moving its way from desktop and server computers into handheld devices, such as smartphones and portable game consoles. While the GPUs inside mobile devices and desktop computers have similar high-level functionality, there are many differences under the hood. For example, a GPU inside a smartphone is usually integrated in a single chip with CPU, DSP, and other application-specific accelerators (e.g. [4]). Instead of having its own graphics memory, an embedded GPU shares the system bus with other computing components to access the external memory and therefore has much less available bus bandwidth than those of laptop and high-performance desktop systems [5]. Also, the only available APIs for current mobile GPUs are OpenGL ES [6], which is a graphics API and does not provide some essential components of GPGPU, such as "scatter" (i.e. write to an arbitrary memory location) and thread-level synchronization. As most existing CPU-GPU optimizations are based on, and optimized for, desktop and server platforms, it is highly desirable to characterize the mobile CPU-GPU platform and revisit the GPGPU strategies in order to better utilize the computational power of a mobile GPU. A study of comparing the use of a mobile CPU (ARM) and a mobile GPU (PowerVR SGX) for executing an image processing pipeline (adjusting geometry, Gaussian blur, and adjusting color) reports that the mobile GPU achieves 3.58x speedup (8.6 seconds per frame for CPU and 2.4 seconds per frame for GPU) [7]. Their investigation is conducted with an emulated version of OpenCL embedded profile [8], which is not available on current commodity smartphones and development boards. Also, the target task of their study is low-level image processing, not high-level vision tasks.

Power and energy efficiency is another critical design considerations when design applications on a battery-powered mobile platform. A mobile handheld device is typically limited by a power ceiling of less than 1 watt, while the power ceilings for the desktop processors alone range from 30 to 150 Watts. In addition to explore the utilization of mobile GPU to speedup time-consuming tasks, it is also important to characterize the power consumption of the mobile GPU and CPU. The overall objective of developing an application on a mobile platform should be to optimize the total energy consumption while meeting the real-time constraint.

In this paper, we investigate the computational capability and energy efficiency of current mobile CPU-GPU system for an exemplar computer vision application, automatic face annotation. We use Nvidia's Tegra SoC/platform [9], which is specifically designed for smartphones and tablets, as the target platform in our study. Running the face recognition algorithm on Tegra's CPU, on average, takes 8.5 seconds to detect and recognize a face. When utilizing Tegra's GPU by OpenGL ES 2.0 to offload the most compute-intensive task, face feature extraction, in the face recognition pipeline, the execution time as well as the total energy consumption can be significantly reduced. This paper is organized as follow: In Section 2 we first review the recent research of mobile computer

vision and energy efficiency of desktop CPU-GPU systems. Section 3 provides an overview of the face annotation system and its runtime profile. Then in Section 4, we show the experimental setup of our study. The experimental results are presented in Section 5. Finally we conclude our study and the future exploration directions.

2 Related works

While state-of-the-art face recognition algorithms can achieve a high accuracy to support automatic face annotation, their implementations on an embedded platform cannot achieve real-time performance due to the demanding computational requirement. Applications targeted for mobile platforms usually remove the compute-expensive operations, or rely on the clouds to do most of the computation. For example, a real-time face annotation system on PDA was demonstrated in [10]. Although it achieves real-time performance, the intensity-comparison based method is not sufficiently robust to handle the luminance variation or pose changes. Hence it could not achieve the level of accuracy needed for real-life applications.

As a sophisticated hardware component with massive parallelism, running tasks on a GPU consumes significant power. The Nvidia 8800GTS graphics card is measured consuming 210W before the kernel launches, and 310W while the kernel is running [12]. On the other hand, the CPU employs several advanced low-power design techniques and power management strategies, thus making it more power efficient. The measured standby power and active power of Intel i7 is 33.03W and 102.2W (for one core) respectively [12]. For applications where the GPU can finish the task in a significantly shorter period of time, in comparison with its CPU counterpart, the performance gain results in energy savings as well, making the GPU a preferred choice from both performance and energy points of view. However, when the GPU speedup is not as pronounced, the choice becomes less obvious. The cost/performance investigation of an Intel Core 2 Duo CPU and a Nvidia CUDA enabled GPU in [11] shows, despite an increase in total system power, using a GPU is more energy efficient when the performance improvement is 5x or greater.

3 Face recognition system

Face recognition enables easy sharing and better management of digital photos and videos. Fig. 1 shows an exemplar face annotation application on smartphones. Given a newly taken photo, or one from the photo gallery in a smartphone, the face regions are identified, recognized and tagged with names automatically. The tagged face(s) could be added to the face database, linked to the user's address book, and/or uploaded the annotated photo to a photo-sharing or social networking websites such as Picasa or Facebook in real-time using the smartphone's Wi-Fi or 3G network connectivity. The face recognition process

can be divided into four steps: (1) Face detection, which scans the whole image to identify face regions. (2) Face landmark localization, which identifies the face landmark locations such as eyes, nose and mouth within a detected face region, and then resize and register the face region accordingly. (3) Face feature extraction, which represents a face region by its features that are invariant or robust to the variations of illumination, pose, expression, and occlusion. (4) Face feature classification, which compares the face feature to the training face set and assigns a name of the most similar identity to the query face.

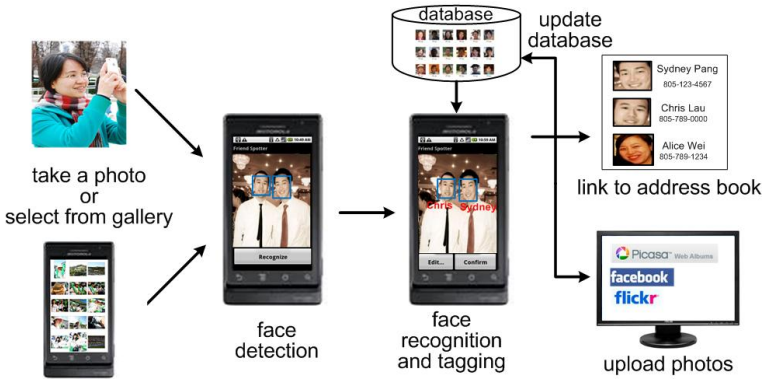

Fig. 1. A face annotation system on smartphones

3.1 Gabor-based face feature extraction

The Gabor-based feature descriptor [13] has been demonstrated as one of the most suitable local representation for face recognition. The Gabor wavelet representation of an image is the convolution of the image with a family of Gabor kernels as defined in the following:

$$\Psi_{\mu,v}(z) = \frac{||k_{\mu,v}||}{\sigma^2} e^{(-||k_{\mu,v}||^2||z||^2/2\sigma^2)}[e^{ik_{\mu,v}z} - e^{-\sigma^2/2}] \quad (1)$$

Where μ and v define the orientation and scale of the Gabor filters, $z = (x, y)$, $||.||$ denotes the norm operator, and the wave vector $k_{\mu,v} = k_v e^{i\phi_\mu}$, where $k_v = k_{max}/f^v$ gives the frequency, $\phi_\mu = \mu\pi/8$ gives the orientation. This representation captures the local structure corresponding to spatial frequency (scale), spatial localization, and orientation selectivity. As a result, it is robust to illumination and facial expression variations. A typical Gabor-based face descriptor uses 40 different Gabor kernels which include 5 different scales and 8 different orientations. After that, it is further processed by Principle Component Analysis (PCA) and Linear Discriminant Analysis (LDA) to reduce its dimensionality and forms the final face feature descriptor. The use of Gabor feature combined with

the PCA-LDA method is reported to achieve 93.83% accuracy on the traditional face recognition dataset FERET [14] and 71.69% on a more challenging photo dataset LFW [15].

3.2 System profiling

In order to understand the complexity of the face recognition system and the performance of such application on modern smartphones, we have implemented a Gabor-based face recognition system on an Android-powered [16] smartphone platform. The details of this platform are revealed in the next section. In our system, we use Android facedetector API [17] to identify face regions in a given photo, and an AdaBoost-based eye localization method to identify the landmark regions. Face feature classification is performed by the K-Nearest-Neighbor method. All the tasks, except the face detection (which is an Android API and the implementation details are not easy to obtained), are implemented in C and compiled using the tool chain provided by Android Native Development Kit r3 (NDK).

Table 1 shows the execution time breakdown of face recognition system running on a 1GHz ARM Cortex A9 CPU. The picture size in this study is 480x1000. The total number of training images is 15. The identified face region is aligned and scaled to 64x80 before extract face feature. The profiling results show that, feature extraction is the most time-consuming part. It takes 6.1 seconds to process one face, which is about 71.8% of the total computing time. Without any optimization, the overall execution time of the face recognition too long to be considered as a real-time mobile applications.

Table 1. Execution time breakdown of face recognition system running on Tegra platform's CPU

Task	Time (sec)	%
Face detection	1.5	17.6
Landmark detection	0.7	8.2
Feature extraction: Gabor wavelet	5.1	60.0
Feature extraction: PCA-LDA	1.0	11.8
Feature classification	0.2	2.4
Total	8.5	100

Since the Gabor wavelet in the face feature extraction is the most dominant component, optimization of this aspect will result in the most improvement. In the following, we examine using the mobile GPU to accelerate this part. The experimental setup is first described in Section 4, and then we discuss the implementation details and the experimental results in Section 5.

4 Experimental setup

Our experiments were performed on a Nvidia Tegra SoC platform with the following specifications: a 1GHz dual-core ARM Cortex A9 CPU, 1GB of RAM, a Nvidia GeForce GPU, and 512MB of Flash memory. This chip is one of the representative heterogeneous processors designed for handheld devices such as smartphones and tablets. A Nvidia Tegra developer kit [9] is available for developing software running on the Tegra chip. Fig. 2 shows our experimental setting: the Tegra board is connected to a VGA monitor and a keyboard and a mouse are also connected. The operating system running on this platform is Android.

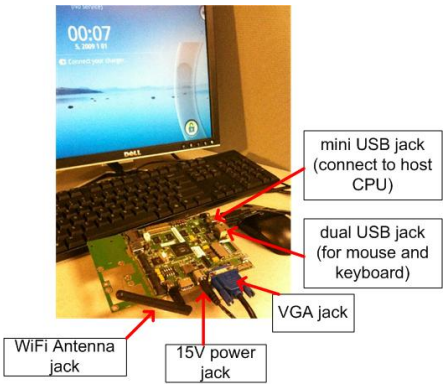

Fig. 2. Tegra development board and the experimental setup

The Tegra board is powered by a 15V DC input which is then converted into 3.3V, 5V, 1.8V and 1.05V for various components on the board by a regulator. It is difficult to precisely measure the power consumption because it requires isolating the traces on the board that provide power to the Tegra chip and measuring the current values. Also, because the CPU and the GPU respectively are integrated in a single chip, it's hard to measure exactly the current drawn by each individual component in the chip. Therefore, we approximate the current used by the CPU and the GPU by measuring the current consumed by the entire board. The average idle current is about 0.2A which is considered as the offset current.

The Tegra GPU has fully programmable unified vertex and fragment shaders. The shaders are programed through OpenGL ES 2.0 [6] which is the primary graphics library for handheld and embedded devices with a programmable GPU. The commonly used high-level API for a desktop environment, such as CUDA or OpenCL, is not supported in this, and any other embedded, platform yet.

5 Execution efficiency and energy efficiency study

Gabor wavelet can be implemented by convolution or the Fast Fourier Transform (FFT). The GPU implementation of the convolution method is suitable only for small-size kernels due to the memory limitation. However, a small-size kernel is not realistic for object and pattern recognition [18]. Therefore, our GPU-based Gabor face feature extraction is based on the FFT method: first transforming both face image and the Gabor kernel into the Fourier space, multiplying them together, and then inverse-transforming the result back to the space domain. In our study, we first take FFT as a benchmark program to investigate the computational and energy efficiency of mobile GPU, and then extend the result to the Gabor face feature extraction processing.

5.1 FFT benchmark on mobile CPU and GPU

Fig. 3 shows the pseudo code of our FFT benchmark program. The CPU implementation is written in C. In the GPU implementation, the GPU shaders, which are launched by the host CPU, perform FFT and IFFT computation. The data needs to first transfer from the CPU domain to the GPU domain and then transfer back after the GPU finishes its computation. The shader program is compiled on the Tegra GPU. After the execution is completed, the resulting image is displayed on the screen.

CPU-implementation	GPU-implementation
1. Allocate memory 2. Generate N*N sample 3. Repeat T times 4. { 5. FFT 6. IFFT 7. } 8. Display result on screen	1. Compile shader program 2. CPU allocate memory 3. CPU generate N*N sample 4. CPU copy data to GPU 5. Repeat T times 6. { 7. FFT 8. IFFT 9. } 10. CPU read data from GPU 11. Display result on screen

Fig. 3. FFT benchmark used in our study

A GPU-acceleration method in [21] is used in our study. Although some other GPU-accelerated FFT methods have been proposed [19][20][22], they were proposed for dedicated hardware and could not be applied to an embedded GPU which has significantly less resources due to the power constraint. The approach used in our study relies on the fragment shader to do the per-pixel (i.e. each sample of the 2D array) computation. The input index and weighting factor

which are used for the calculation of the each sample are pre-computed and stored in the texture memory.

We ran a FFT benchmark program, which performs FFT and IFFT 50 times, on Tegra's CPU and GPU respectively for comparison. The measured power consumption results are shown in Figure 5. Both CPU and GPU start running roughly at reference time 0.6 second. After CPU starts running, it takes about 0.4 second for the CPU to initialize the GPU and to transfer data from the CPU to the GPU before the GPU runs at its full capacity. After the FFT and the IFFT computations are completed, the application program is still running (but does nothing). Therefore, the power consumption level is still higher than the level of the idle stage before the application was launched.

The measurement results show that, for this FFT benchmark, the GPU is 3x faster and consumes 8% more power than the CPU (1 second vs. 3.1 seconds, and 4.0 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.

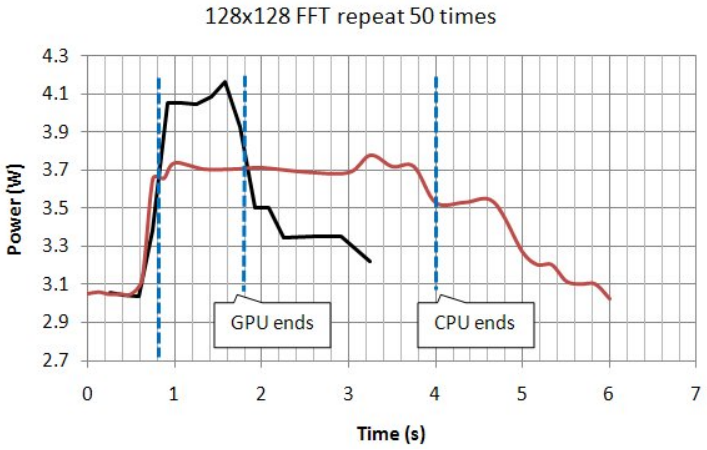

Fig. 4. Power consumption of our FFT benchmark on mobile CPU and GPU

5.2 GPU accelerated Gabor-based face feature extraction

The FFT benchmark result demonstrates that using a mobile GPU is not only more computationally efficient but also more energy efficient. We then extend the GPU-accelerated FFT and IFFT to compute the Gabor-based face feature extraction, and compare the results with the CPU implementation. The left side of Fig. 5 shows the pseudo code of the Gabor-based face feature extraction. In the GPU implementation, the FFT and IFFT are the same shader program

with different input arguments in order to perform either forward or inverse transform, and the MULTIPLY is a separate shader program. In other words, different shader programs have to be swapped back and forth repeatedly to complete the task. The execution time of both CPU and GPU implementation are shown in the first two rows of Table 2. The GPU implementation runs 4.25x faster than the CPU implementation (1.2 seconds vs. 5.1 seconds) while consuming slightly more power (3.75 Watt vs. 3.52 Watt).

Since the computation of convolving 40 different Gabor kernels with a face image can be computed concurrently and independently, processing multiple Gabor kernels in a batch mode may further improve the performance by reducing overall time spent on swapping shader programs. Three different configurations are examined in our study: The first configuration (Fig. 6(a)) is the original method which performs 40 Gabor wavelets with 40 different kernels. The second configuration (Fig. 6 (b)) combines four kernels in a batch, and the shader program is configured to draw a 256x256 quad but performs four 128x128 FFTs while each FFT tile has a different texture access address. This could be easily performed by loading another texture to lookup the index. The third configuration (Fig. 6 (c)), similar to the second one, combines nine Gabor kernels in a batch and performs nine 128x128 FFTs at a time.

Single mode	**Batch mode**
1. FFT (face)	1. FFT (N tile face) // same face in each tile
2. for i = 1~40	2. for i = 1~ceil(40/N)
3. {	3. {
4. FFT (kernel i)	4. FFT (N tilt kernel) // different kernel in each tile
5. MULTIPLY ()	5. MULTIPLY()
6. IFFT()	6. IFFT()
7. Read results back to CPU	7. Read results back to CPU
8. }	8. }

Fig. 5. Doing 40 Gabor filter in the single mode and batch mode.

The measurement results show that, however, the batch mode does not reduce the computation time as we expected. As shown in Table 2, GPU_1x1, GPU_2x2, and GPU_3x3 take 1.2, 1.4, and 1.5 seconds respectively to complete the assigned tasks. This could be explained that a larger amount of data is required for the computation when running a larger number of concurrent tasks. If the GPU cache size is not large enough, it takes more time to store and load data from the main memory.

5.3 Overall performance

Computing Gabor representation of a face image is the most time consuming part of the whole face recognition system. It takes about 5.1 seconds for a 1 GHz ARM processor to complete this task. The mobile GPU takes only 1.2

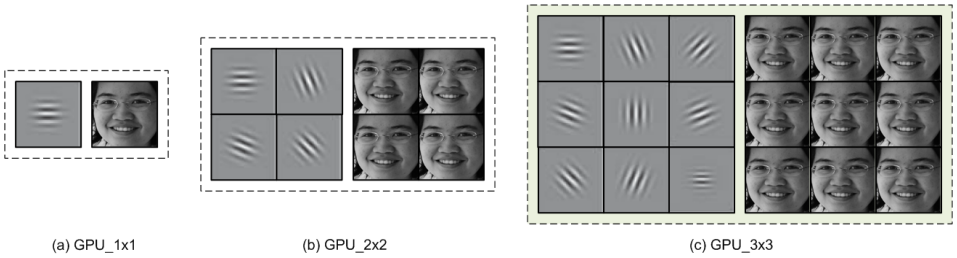

Fig. 6. Combine various number of Gabor kernels to perform larger size FFT together. (a) Perform a 128x128 FFT for one kernel at a time. (b) Perform four 128x128 FFTs for four kernels at a time. (c) Perform nine 128x128 FFT for nine kernels at a time.

Table 2. The execution time and energy consumption of different implementation configurations

configuration	# of batch	Time (sec)	Power (Watt)	Energy (J)
CPU (1x1)	40	5.1	3.52	17.95
GPU (1x1)	40	1.2	3.75	4.50
GPU (2x2)	10	1.4	3.59	5.02
GPU (3x3)	5	1.5	3.63	5.44

seconds to complete the same task, which represents a 4.25x speedup. As shown in Fig. 7, with the GPU successfully offloading 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 16.3 J while the CPU only implementation consumes 29.8 J. 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 in the future the opportunity of utilizing mobile GPU to remove these new computational bottlenecks to achieve better performance and energy consumption level.

6 Conclusion and future works

In this paper, we investigate the computing power and energy consumption of a mobile CPU-GPU platform for mobile computer vision applications. Compared to a CPU-only implementation, our preliminary GPU-accelerated Gabor face feature extraction, the most compute-intensive task in a face annotation system can achieve a 4.25x speedup and 3.94x reduction in energy consumption. This experimental investigation confirms that a mobile GPU, although is 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. Therefore,

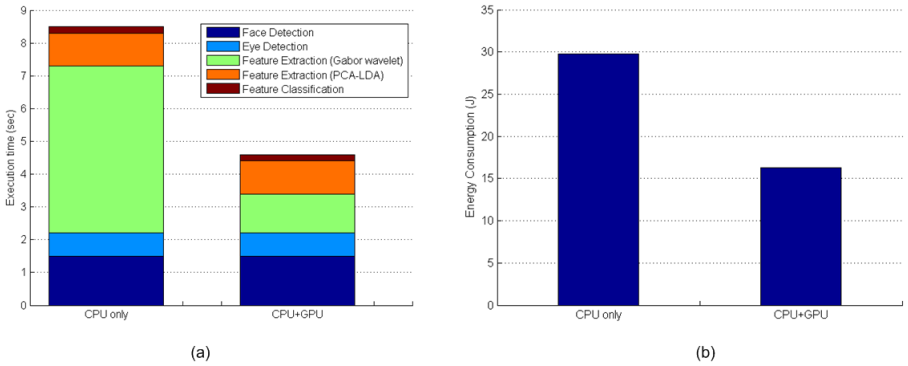

Fig. 7. Comparison of face recognition system running on Tegra's CPU and CPU+GPU. (a) Execution time (b) Total energy consumption

the performance improvement achieved by GPU-based computing also results in overall energy reduction, which is a tremendous benefit for mobile devices.

Due to the lack of a higher level programming environment, such as CUDA, for mobile GPUs, it is difficult to port existing GPU-optimized algorithms to the mobile SoCs, even for those already designed for generic GPU architectures. It is worthwhile to further explore low-power architectures and extend them toward a more programmable, general-purpose architecture. While increasing the programmability may somewhat compromise the execution efficiency or increase the power consumption, exploring the tradeoffs between energy efficiency and the programmability and identifying a solution for easier programming without costing too much degradation in performance and energy consumption are necessary steps for improving the productivity for programming for mobile GPUs.

References

1. Nvidia CUDA Compute Unified Device Architecture Programming Guide; Version 2.0, Nvidia Corporation, www.nvidia.com,2008
2. Samant, SS., Xia, J., Muyan-Ozcelik, P., Owens, J.D.: High performance computing for deformable image registration: Towards a new paradigm in adaptive radiotherapy. Medical Phisics, 2008.
3. Kim, J-S., Hwangbo, M., Kanade, T.: Realtime Affine-photometric KLT Feature Tracker on GPU in CUDA Framework, IEEE Workshop of Embedded Computer Vision 2009
4. OMAP3 family of multimedia application processors, Texas Instruments Inc., http://focus.ti.com, 2007
5. Akenine-Moller, T.; Strom, J.: Graphics Processing Units for Handhelds Proceedings of the IEEE, Volume 96 , Issue 5, 2008 , 779-789
6. Munshi, A., Ginsburg, D., and Shreiner D.,: OpenGL ES 2.0 Programming Guide, Addison-Wesley, USA, July 2008.

7. Leskela, J.; Nikula, J.; Salmela, M.: OpenCL embedded profile prototype in mobile device IEEE Workshop on Signal Processing Systems, (SiPS 2009). Page(s): 279-284
8. Khronos OpenCL Working Group, A. Munshi Ed., The OpenCL Specification, , Version 1.0, Rev. 43, Khronos Group, USA, May 2009.
9. Nvidia Corporation, http://tegradeveloper.nvidia.com/tegra/
10. Chu, S-W., Yeh, M-C., Cheng, K-T.: A real-time, embedded face-annotation system. ACM MM Technical demonstrations (2008)
11. Rofouei, M., Stathopoulos, T., Ryffel, S., Kaiser W., Sarrafzadeh M.: Energy-Aware High Performance Computing with Graphic Processing Units. Workshop on Power Aware Computing and Systems (HotPower 2008). San Diego. December 8-10.
12. Ren, D.Q., Suda, R.: Power Efficient Large Matrices Multiplication by Load Scheduling on Multi-core and GPU Platform with CUDA International Conference on Computational Science and Engineering, (CSE 2009).
13. Su, Y., Shan, S., Chen, X., Gao, W.: Hierarchical Ensemble of Global and Local Classifiers for Face Recognition. In IEEE Transactions on Image Processing (2009), vol. 18, issue 8, 1885-1896
14. Phillips, P.J., Moon, H., Rizvi, S.A., Rauss, P.J., The FERET evaluation methodology for face-recognition algorithms. PAMI,22(10), pp. 1090-1104, 2000.
15. Huang, G.B., Ramesh, M., Berg, T., Learned-Miller, E., Labeled faces in the wild: a database for studying face recognition in unconstrained environments. University of Massachusetts, Amherst,Technical Report 07-49, 2007.
16. http://developer.android.com/index.html
17. http://developer.android.com/reference/android/media/FaceDetector.html
18. Fialka, O., Cadik, M., FFT and Convolution Performance in Image Filtering on GPU, Tenth International Conference on Information Visualization, 2006
19. Nvidia Corp., CUDA CUFFT Library
20. Mitchell, J.L., Ansari, M.Y. , Hart, E.: Advanced image processing with DirectX 9 pixel shaders. in ShaderX2: Shader Programming Tips and Tricks with DirectX9.0, Wolfgang Engel, Ed. Wordware Publishing, Inc., 2003.
21. Sumanaweera, T., Liu,D.,: Medical image reconstruction with the FFT in GPU Gems 2, Matt Pharr, Ed., pp.765-784. Addison-Wesley, 2005.
22. Brandon, L.D.; Boyd, C.; Govindaraju, N.: Fast computation of general Fourier Transforms on GPUS IEEE International Conference on Multimedia and Expo (ICME 2008) 5 - 8