GPU based Parallel Image Processing Library for Embedded Systems

Mustafa Cavus, Hakkı Doganer Sumerkan, Osman Seckin Simsek, Hasan Hassan,
Abdullah Giray Yaglikci and Oguz Ergin
TOBB University of Economics and Technology, Ankara, Turkey

Keywords: GPGPU, OpenCL, Embedded Systems, Image Processing, Parallel Computing.

Abstract: Embedded image processing systems have many challenges, due to large computational requirements and other physical, power, and environmental constraints. However recent contemporary mobile devices include a graphical processing unit (GPU) in order to offer better use interface in terms of graphics. Some of these embedded GPUs also support OpenCL which allows the use of computation capacity of embedded GPUs for general purpose computing. Within this OpenCL support, challenges of image processing in embedded systems become easier to handle. In this paper, we present a new OpenCL-based image processing library, named TRABZ-10, which is specifically designed to run on an embedded platform. Our results show that the functions of TRABZ-10 show 7X speedup on embedded platform over the functions of OpenCV on average.

1 INTRODUCTION

Today's GPUs can use hundreds of parallel processor cores and can execute thousands of parallel threads. Their performance exceeds the performance of CPUs in arithmetic throughput and memory bandwidth. In this trend, OpenCL aims to make programming many core GPUs easier for programmers. OpenCL is a platform independent open standard which is able to run on different platforms of different vendors. OpenCL supports the use of many computation resources like embedded GPUs, CPUs, FPGAs or any other OpenCL enabled computation units.

OpenCL fits well for a wide spectrum of applications and one of the common usage area is image/video processing. One of the most important challenges in the field is the high computation requirement to achieve high accuracy and real-time processing. With increasing video resolution and data frame sizes, it is getting harder to achieve real-time performance.

GPGPU offers the designers an option of high data-parallel computation. Most of the image processing algorithms fit well on this approach. Also, they can benefit the single instruction multiple data (SIMD) architecture of GPUs and can be effectively parallelized. However, it is not possible to say that all image processing algorithms fit well with data parallel approach, and they cannot be ported to GPUs with significant speedups. Although image processing algorithms are well suited to massively parallel architecture of GPUs, some of them are failed to achieve performance speedup. In addition to all of these, embedded architecture limitations give additional restrictions and this makes harder to port certain functions to GPU.

In this paper we present an image processing library that is compatible with embedded devices. We chose OpenCV as a reference model and compared our library both in functionality and performance with OpenCV, which is one of the most popular and efficient image processing toolkits. To create a library that is suitable for embedded platforms and able to compete with OpenCV, we implemented key algorithms of image processing. While implementing these algorithms, our main purpose was to accelerate these algorithms on GPGPU using OpenCL.

2 RELATED WORK

The most common applications on GPGPU are image processing applications. Image processing operations perform the same computation on many pixels; they can exploit the massive data parallelism and outperform their CPU implementations.

The starting point of this research area was not based on GPGPUs, but based on regular GPUs on embedded platforms. First approaches include using OpenGL and DirectX APIs to program GPU's shader cores and programmable pipeline using GLSL (Khronos Group, 2012) (Khronos Group, 2004). Especially using OpenGL ES 2.0 on embedded platforms to increase performance of the image processing algorithms was a hot topic. The invention of CUDA and OpenCL maximized the use of GPGPUs for image processing and other applications.

2.1 GPU Model

General-purpose GPU model was achieved using programmable shader infrastructure. GLSL is the shader programming language which enables implementing not only graphics shader code, but also general purpose computation. A survey of general-purpose computation on GPUs is described in (Owens, J. 2007).

Singhal et al. implemented many image processing, color conversion and transformation algorithms and applications like Harris corner detection and real-time video scaling for handheld devices (Singhal, N. 2010). They used OpenGL ES 2.0 (OpenGL for Embedded Systems) (Munshi, A. 2008) and its texture hardware to achieve their goal.

There are other open source projects like GPUCV (Farrugia, J-P. 2006), MinGPU (Babenko, P. 2008) and OpenVIDIA (Fung, J. 2005). While GPUCV and MinGPU support using GPU and GPGPU functionalities simultaneously, OpenVIDIA can only use CUDA and implements a set of useful image processing functions.

2.2 GPGPU Model

Changing the computation and programming capabilities of GPUs, NVidia developed Compute Unified Device Architecture (CUDA) (Lindholm, E. 2008) and made their GPUs, GPGPUs. GPGPUs support Single Instruction Multiple Thread (SIMT) architecture. SIMT architecture allows one instruction used by many threads and these threads can process multiple data, resulting in excessive performance gain. GPGPUs have been used extensively to exploit this performance gain in image processing applications. While some papers implement image processing functions from scratch (Yang, Z. 2008), some of them are implemented over existing libraries (Farrugia, J-P. 2006) (Kong, J. 2010). There is also some research on particular image processing algorithms. In (Luo, Y. 2008), Luo et al. used CUDA to enhance the speed of canny edge detection algorithm.

OpenCL is the standardized version of general purpose computing platform (Khronos OpenCL Working Group. 2008) (Munshi, A. 2011). Unlike CUDA, OpenCL works on various processors including embedded processors, DSPs and FPGAs (Czajkowski, T. 2012). OpenCL and CUDA have little differences in coding and capability as shown in (Karimi, K. 2010) (Fang, J. 2011) (Du, P. 2012).

3 OpenCL EMBEDDED PROFILE

3.1 OpenCL Programming Model

OpenCL is an API which enables heterogeneous parallel programming on various devices like GPUs, DSPs, FPGAs and even CPUs. OpenCL specification is maintained by Khronos Group, but many vendors contribute to the improvement of this specification (Khronos OpenCL Working Group. 2008). OpenCL programming model consists of one host and one or more compute devices as shown in Figure 1.

Host part runs the C/C++ part of the code (OpenCL supports both C and C++), and communicates with the compute devices. The code that runs on compute units is called kernel and written in OpenCL C language, which is a subset of regular C language with functional extensions. Kernels can be compiled just-in-time or the program can read the pre-compiled device specific binary and execute this binary on compute units. These kernels are executed as single program multiple data which are grouped by 1D, 2D or 3D set of work items. Work items are grouped together into work groups. Work items can be executed in a single instruction multiple data (SIMD) fashion. The processors hierarchy and memory hierarchy is shown in Fig. 2.

There can be multiple multiprocessor infrastructures which have multiple processors. In Figure 2, SP means streaming processor which is the building block of compute device. SPs are grouped into Multiprocessors.

Multiprocessors have their own register files and their own on chip cache which is called local memory in OpenCL and shared memory in CUDA. Device memory is the global memory of the compute unit that consists of RAM of the device.

Device memory may include specific parts called texture cache and constant cache. These memory types are not mandatory, but if they exist,

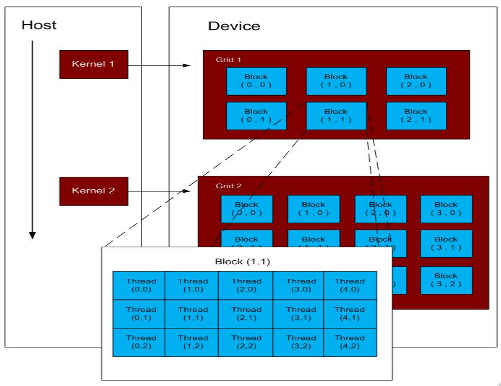

Figure 1: GPGPU Programming Model.

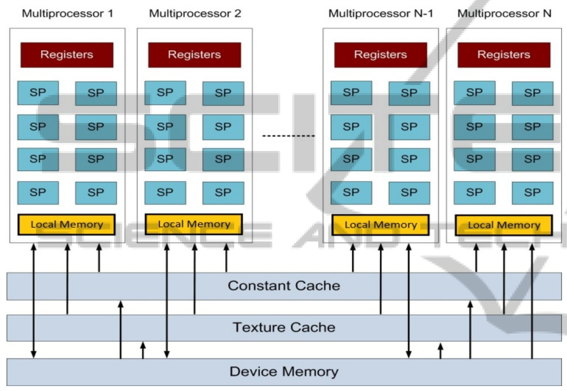

Figure 2: GPGPU Processor and Memory Hierarchy.

they are extremely useful.

3.2 OpenCL Programming on Embedded Profile

Unlike other Khronos standards, OpenCL does not have an ES (embedded systems) specification, but it has embedded profile which is a subset of OpenCL specification (Khronos OpenCL Working Group. 2008). Embedded profile for example, does not have to support double, long or half types, rounding modes can be moderate like round to zero, 3D image support is not mandatory and such.

Even though OpenCL has an embedded profile specification, there were not available devices or manufacturers until recently. There was some effort to be able to implement embedded profiled OpenCL, but it was not as efficient as it should be without the hardware support (Leskela, 2009). Using embedded system GPUs which support OpenGL ES 2.0 was the main idea among these researches. OpenGL ES 2.0 enables programmable shaders and with OpenGL interface, useful functions can be implemented with extensive effort (Cheng, K. 2011) Lopez, M. (2011).

4 TRABZ-10 IMAGE PROCESSING LIBRARY

TRABZ-10 is an open source image processing library for embedded platforms with OpenCL support. Development is done in both Linux and Windows environment but testing of the functions is done using Vivante GC2000 GPU on Freescale i.Mx6q sample board. Vivante GC2000 board is considered a base and implementation and optimizations are done for this particular product.

Even though this product is very powerful and efficient, it has downsides because of the embedded profile constrictions. There are little small size of local memory on the device and very limited number of registers. Local memory is not on chip, but a part of global memory, so using it does not increase performance. Also it has no usable texture memory and support for OpenCL images is limited.

Considering these bottlenecks, we developed an image processing library for embedded systems. The comparisons of the implemented functions are done using OpenCV.

TRABZ-10 image processing library functions are grouped in 6 major sections. These are Matrix operations, filtering, morphology, transformation, feature detection and conversion.

4.1 Matrix Operations

In matrix operations section there exist the basic matrix operations such as matrix addition, subtraction, multiplication, inversion, etc. All of the matrix operation functions take advantage of GPU, except a small set. Min, max functions are implemented on CPU side due to considerable amount of branching. These algorithms also implemented by reduction but the performance results showed no gain over OpenCV.

4.2 Filtering Functions

Filtering functions refer to a class of algorithms that are mostly about the convolution operations with different filters. In this group we have implemented 11 functions including laplacian, sobel, scharr, median, bilateral, gaussian, box, normalized box and blur filters. These filters are the most frequently used filters in image processing.

Image filtering is one of the most considerable fields for OpenCL because many of the image filtering algorithms is the perfect fit for parallel processing in memory access patterns and mathematical complexity. To utilize the benefits, especially the massively parallel nature of OpenCL, best way is to divide image into blocks and assign these blocks to threads. Therefore, each thread is responsible for accessing global memory, copying pixel data to shared memory if it is needed and computing the wanted result for a single pixel.

4.3 Morphological Operations

Morphological operations are a subset of image filtering. The difference between regular filters and morphological operations is, filters multiply and add the values in the filter range and finds out the necessary pixel value, while morphological operations use widely known "fit", "hit" and "miss" functions over a structuring element.

The main functions of morphology are erosion and dilation. Erosion is the intersection between image and structuring element and dilation is the union between them. These functions are more difficult to implement than filter functions due to the comparison and branching. Since morphological operations are applied on binary images, performance loss due to the branching can be compensated.

4.4 Image Transformation

In this section there exist four image transform functions which convert an image from one domain to another. One of these functions is DCT (discrete cosine transform) and one other function is DFT (discrete Fourier transform). The other two functions are inverse DCT and inverse DFT.

Discrete cosine transform function calculates a matrix which is represented as a sum of sinusoids of varying magnitudes and frequencies from the given image with the shown in formula (1).

$$B_{pq} = \alpha_p \alpha_q \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} A_{mn} \cos\frac{\pi(2m+1)p}{2M} \cos\frac{\pi(2n+1)q}{2N}, \quad \begin{matrix} 0 \le p \le M-1 \\ 0 \le q \le N-1 \end{matrix}$$
$$\alpha_p = \begin{cases} 1/\sqrt{M}, & p=0 \\ \sqrt{2/M}, & 1 \le p \le M-1 \end{cases} \quad \alpha_q = \begin{cases} 1/\sqrt{N}, & q=0 \\ \sqrt{2/N}, & 1 \le q \le N-1 \end{cases} \tag{1}$$

Discrete Fourier transform function calculates a matrix which is represented as a sum of complex exponentials of varying magnitudes, frequencies, and phases from the given image. Discrete Fourier transform is computed using the FFT (fast Fourier transform) algorithm which is a fast way to compute DFT. Mathematical representation of FFT is shown in formula (2).

$$F(p,q) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n) e^{-j2\pi pm/M} e^{-j2\pi qn/N} \quad \begin{matrix} p = 0,\ 1,\ \dots,\ M-1 \\ q = 0,\ 1,\ \dots,\ N-1 \end{matrix} \tag{2}$$

4.5 Geometric Transformations

This section consists of nine geometrical transformation functions which perform a deformation operation on the pixel grid, and map the deformed grid to the destination image. This mapping is done from destination to source in order to avoid sampling artifacts. It means that instead of computing destination coordinates for each pixel in the source image, source coordinates are computed for each pixel in the destination image. In this way, every pixel in the destination image is covered and each of them is visited only once.

4.6 Color Conversion

This group of functions convert input image from one color space to another color space. Color conversion kernels are low complexity kernels with no data dependency. Since different color spaces can be useful for different kind of image processing algorithms and video coding/decoding techniques, they need to be accelerated. They also could be ported to the GPU well, because these conversion functions exhibit significant amount of data parallelism. In TRABZ-10 we have implemented commonly used color conversion kernels such as YUV, HSV, LUV, XYZ to RGB and their reversed conversions. RGB to binary and RGB to grayscale functions are also implemented in this group of functions.

4.7 Feature Detection

In this group of functions we have implemented 6 different algorithms. These algorithms give the features of images as a lines, corners or circles. We have implemented Canny and Hough (Line) transformation to detect edges, Harris, Eigen vector and Eigen value to detect corners and Circle Hough transformation to detect circles of the image.

5 IMPLEMENTATION STRATEGIES

In this section we will discuss about the general OpenCL coding strategies and optimizations. We first implemented the naïve kernels and then tried to vectorize the code. The use of local memory was very limited because Vivante GC2000 has as small as 1 KB of local on-chip memory for general purpose computation. After that point we passed over algorithms to find algorithm dependent bottlenecks and exploit parallelism even further. We also tried connecting kernels to reduce CPU-GPU data transfers and fusing multiple kernels into one kernel.

Increasing the memory bandwidth was the first thing to do to optimize the library. Coalescing memory accesses and avoiding bank conflicts while using shared memory was a high priority optimization strategy.

Vector data types are supported for char, uchar, short, ushort, int, uint, float, long, and ulong in OpenCL. The contribution of using vector sizes to the performance is architecture dependent. Certain GPGPU architectures are more efficient in using vector data types in calculation of the operands and some are not. Since GC2000 has well-working vector units, exploiting this feature increases performance gain.

Since the CPU and GPU are on the same chip in Freescale i.Mx6q processor, data copy between CPU and GPU is not necessary; they access the same off-chip RAM. This strategy is also used in this library to avoid performance degradation caused by memory copy operation.

The last strategy in our work was connecting different kernels. We developed a framework that connects different kernels and reduces the CPU-GPU data transfers. Every kernel output is given to the input of another one, so data transfer and system call overhead can be discarded. As a future work, we will apply this framework for whole library.

Kernel connection gives an average of 3x speed up over sequential kernel execution as can be seen in Figure 3.

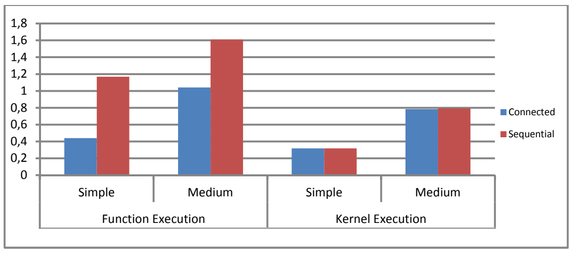

Figure 1: Simple and Medium Kernels.

6 LIMITATIONS OF EMBEDDED PROFILE

There are three major problems we encountered with embedded profile; small register size, small local memory and limited instruction memory. GC2000 has 64 32-bit registers on each core, 1 KB of local memory and it has a limit of 512 instructions per kernel. Since it has these memory limitations, using registers and local memory is really challenging because of the register spill. When registers are not enough, each core holds the register data on global memory, which is off-chip and really slow, and this degrades performance. In GPGPU model model each core hosts a large number of thread groups and every thread group accesses memory. Between these accesses, thread groups wait for the completion of the previous memory access. Between these waits, scheduler switches to another active thread group. So this is why limited register size also limits the total active thread group count and this limits the scheduler's efficiency of memory latencies.

The other major problem was limited instruction memory. Since embedded GPUs have a limited instruction memory, larger kernels can exceed the instruction limit. GC2000 has an instruction limit of 512 per kernel and this prevents implementing complex and large algorithms. In our library the most challenging algorithm is FFT and the limited instruction memory of GC2000 is one of the reasons behind this challenge.

7 RESULTS

This section covers the experimental results for transformation, filtering and matrix functions. Our test environment is an embedded platform which has a Quad Core ARM Cortex A9 CPU and Vivante GC2000 GPU. Specifications of the environment can be seen in Table 1.

Table 1: Embedded Platform Specifications.

Embedded Platform					
GPU Type	GPU Clock	GPU Cores	CPU Type	CPU Clock	Memory Clock
GC2000	600 Mhz	4	Cortex A9	1200 Mhz	533 Mhz

All of the results that are shown on the figures are kernel execution times in seconds. OpenCV functions are executed on ARM Cortex A9 and TRABZ-10 results are from Vivante GC2000 GPGPU. As it is seen from the figure, TRABZ-10 has better results in all filtering functions with different image sizes than OpenCV. It can also be seen that speed up of functions increases with bigger image sizes.

Filtering function results showed that TRABZ-10 is up to 7X faster than OpenCV on our embedded platform. Figure 4 shows the achieved speedups against OpenCV in filtering functions.

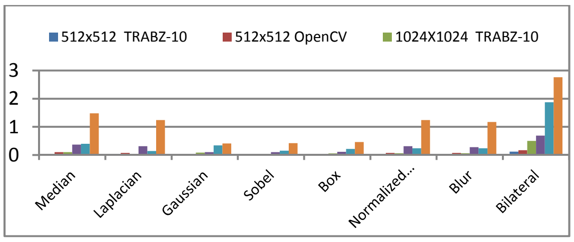

Figure 2: Execution Times of Filtering Functions with Different Image Sizes on Embedded Platform.

Transformation functions results showed that TRABZ-10 is up to 8x times faster than OpenCV on our embedded platform. Figure 5 shows the execution times on our embedded platform.

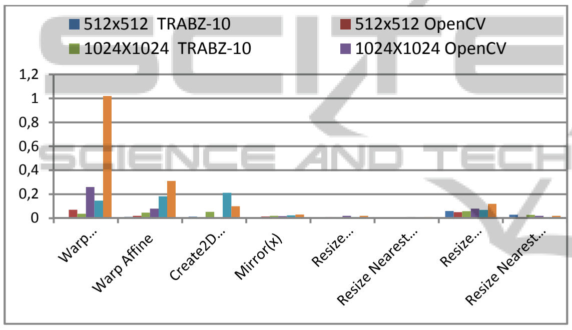

Figure 3: Execution Times of Transformation Functions with Different Image Sizes on Embedded Platform.

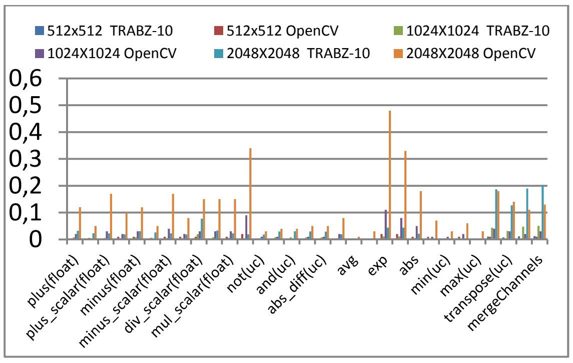

Figure 4: Execution Times of Matrix Operations on Embedded Platform.

As the figure shows the most gained speed-up is over Warp Perspective and Warp Affine functions. On other functions execution times are relatively close to the OpenCV execution times.

Figure 6 shows the execution times of another important set of operations, matrix operations, which is used widely on image processing. From the figure, it can be seen that TRABZ-10 has a better performance on almost all of the functions. It also has a better average speed up than filtering and transformation function groups. Since some of the matrix functions are recursive by nature, these functions are implemented on CPU side.

8 SAMPLE APPLICATION: FACE DETECTION

To test our library, we implemented a naive face detection application. To be able to compare the results, we both implemented an OpenCV version and TRABZ-10 version with the same algorithm.

The proposed method is based on color and feature-based face detection. The algorithm can detect human face under different lighting conditions with high speed and high detection ratio. The first step of the implementation consists of average luminance calculation in YCbCr color space to determine the compensated image. Chrominance (Cr) is used to detect pixels containing human skin color. To remove high frequency noise, a low pass filter is applied. Separate human-skin regions are labeled and corner points of each region is stored. Regions that do not pass the height to width ratio are eliminated. Then mouth detection is performed using vertical based histogram for matrix (M) given in formula (4). If mouth is not detected in region, it is eliminated. Similar operation is done for eye detection by determining vertical based histogram for pixels whose luminance is slightly darker than average skin-color. Regions that pass all these steps are marked as a human face.

$$\alpha = \cos^{-1}\left(\frac{0.5(2R^{\iota}+G^{\iota}-B)}{\sqrt{(R^{\iota}-G^{\iota})^{2}+(R^{\iota}-B)(G^{\iota}-B)}}\right)$$
$$M_{pq} = \begin{cases} 0, & \alpha < 90 \\ 1, & otherwise \end{cases} \tag{4}$$

An example of detection results are presented in the Figure 7. Subjects considered in selected images belong to several groups and lighting conditions. A detected face is a correct detection with a small amount of about 5% tolerance.

The detection ratio is computed by the ratio of the number of correct detections in a gallery to that of all the human faces in the gallery. The detection rate is around 95%, if the images considered have faces camouflaged with the background then the face detection efficiency comes down to around 90%. The reason for this decrease in the detection rate is due to the reason that the face is merged with that of the background.

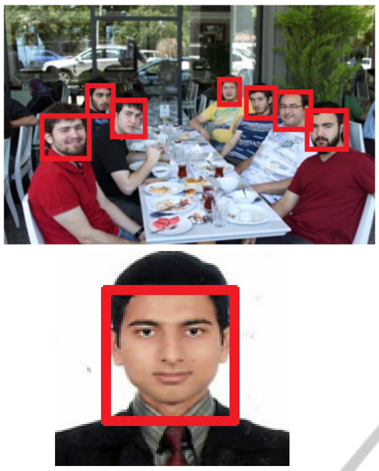

Figure 7: Results of face detection with TRABZ-10.

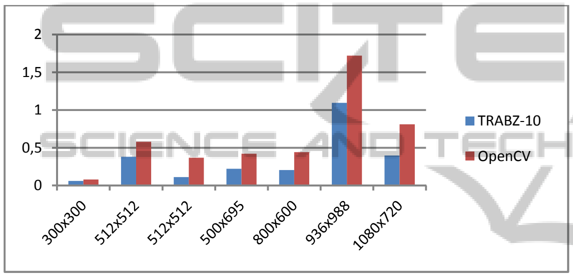

Figure 8: Performance results of face detection.

On the performance side, TRABZ-10 has better performance for different image sizes. As it can be seen from the Figure 8, there are two different results for image size 512x512. The reason behind this difference is the number of features on these images. Number of features on the image is the main factor that affects the run time.

9 CONCLUSIONS

In this paper, we presented our open-source image processing library which is implemented using OpenCL. We developed base functionalities including filter functions, transformations, conversion and various matrix operations. Our results show that, TRABZ-10 has speedups for most of its functions compared to the embedded version of OpenCV. This library constitutes as a start point of image processing on embedded GPGPUs. It has basic functionality for now, but it is open for improvement. As a future work, we are planning to get power results and test our library on different embedded profiles that has OpenCL support and adding more functionality to the library.

REFERENCES

Khronos Group. (2012). *OpenGL Shading Language Specification.* Available: http://www.opengl.org/documentation/glsl. Last accessed 25th April 2013.

Khronos Group. (2004). *OpenGL ES 2.0 Specification. Available*: http://www.khronos.org/opengles. Last accessed 25th April 2013.

Munshi, A (2008). OpenGL ES 2.0 *Programming Guide*. Addison-Wesley Professional.

R. Marroquim, A. Maximo, Introduction to GPU Programming with GLSL. Computer Graphics and Image Processing (SIBGRAPI TUTORIALS), 2009 *Tutorials of the XXII Brazilian Symposium on, vol., no.*, pp.3,16, 11-14 Oct. 2009.

Owens, J. (2007). A Survey of general-purpose computation on graphics hardware. *Computer graphics forum. 26* (1), p80-113.

Singhal, N. (2010). Implementation and optimization of image processing algorithms on handheld gpu. IEEE *International Conference on Image Processing.* p4481-4484.

Farrugia, J-P. (2006). GPUCV: A framework for image processing acceleration with graphics processors. IEEE *International Conference on Multimedia and Expo, 2006.* p585-588.

Babenko, P. (2008). MinGPU: a minimum GPU library for computer vision. *Journal of Real-Time Image Processing. 3* (4), p255-268.

Fung, J. (2005). OpenVIDIA: parallel GPU computer vision. *Proceedings of the 13th annual ACM international conference on Multimedia.* p849-852.

Lindholm, E. (2008). NVIDIA Tesla: *A unified graphics and computing architecture.* Micro, IEEE. 28 (2), p39-55.

Yang, Z. (2008). Parallel image processing based on CUDA. IEEE *International Conference on Computer Science and Software Engineering. 3* (1), p198-201.

Kong, J. (2010). Accelerating MATLAB image processing toolbox functions on GPUs. ACM *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units.* p75-85.

Luo, Y. (2008). Canny edge detection on NVIDIA CUDA. IEEE *Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, CVPRW'08. p1-8.

Khronos OpenCL Working Group. (2008). OpenCL Specification. Available: http://www.khronos.org/registry/cl. Last accessed 25th April 2013.

Czajkowski, T. (2012). From OpenCL to high-performance hardware on FPGAs. IEEE 22nd *International Conference on Field Programmable Logic and Applications* (FPL). p531-534.

Karimi, K. (2010). *A performance comparison of CUDA and OpenCL.* arXiv preprint arXiv:1005.2581.

Fang, J. (2011). A comprehensive performance comparison of CUDA and OpenCL. IEEE *International Conference on. Parallel Processing* (ICPP). p216-225.

Du, P. (2012). From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing. 38* (8), p391-407.

Munshi, A (2011). *OpenCL programming guide*. Addison-Wesley Professional.

Leskela, J. (2009). OpenCL embedded profile prototype in mobile device. IEEE *Workshop on Signal Processing Systems,* (SiPS). p279-284.

Cheng, K. (2011). Using mobile GPU for general-purpose computing–a case study of face recognition on smartphones. IEEE *International Symposium on VLSI Design, Automatin and Test* (VLSI-DAT). p1-4.

Lopez, M. (2011). Accelerating image recognition on mobile devices using GPGPU. *International Society for Optics and Photonics IS&T/SPIE Electronic Imaging. 1* (1), p78720R-78720R-10.