Energy Efficient Object Detection on the Mobile GP-GPU

Fitsum Assamnew Andargie†, Jonathan Rose††, Todd Austin§ , and Valeria Bertacco§

†School of Electrical and Computer Engineering, Addis Ababa University, Addis Ababa
††The Edward Roger Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto
§Computer Science and Engineering Department, University of Michigan, Ann Arbor

***Abstract*— Smartphones and tablets now include General Purpose Graphics Processing Units (GP-GPUs) that can be used for computation beyond driving the high-resolution screens. In this paper we present a mobile GP-GPU-based object detection algorithm and system, based on the work by Viola and Jones (which is also used in the OpenCV library). This implementation achieved twofold speed up compared to OpenCV running on the CPU of the same smartphone, and up to 84% energy saving. Interestingly, the new implementation saves energy vs. the CPU even when it executes slower than the OpenCV implementation, because the GPU consumes less power than the CPU, something that is not typical in desktop or laptop systems.**

Keywords— smartphones; GP GPU; object detection; Adreno GPU;OpenCL

I. INTRODUCTION

Fueled by the exponential advances in semiconductor technology, smartphones have progressed dramatically in the last few years. This trend has enabled an explosion of applications that are targeted for use on smartphones. However, some applications such as computer vision still require more computational power for an acceptable user experience. In addition, any application running on smartphones needs to be aware of its consumption of energy. There are several ways this can be achieved. One way is the development and use of better algorithms while the other is use of the already existing energy efficient co-processors/accelerators found in the smart phones. One example of these co-processors is the graphics-processing unit (GPU).

In recent years, the graphics-processing unit in smart phones has followed suit of its desktop, laptop and server counterparts in becoming capable of being used for general purpose computing. Several vendors of smartphones processors now include general purpose GPUs (known as GP-GPUs). Examples are the Adreno series of GPUs from Qualcomm [1], the Mali series from ARM [2] and the PowerVR Series from Imagination Technologies [3]. These GP-GPUs can be programmed using the Open Computing Language (OpenCL) [4] for general purpose computing and many other graphics libraries for graphics application. The use of OpenCL makes the development of general software on the GPU comparatively easier as compared to previously where general problems had to be cast as graphics problems in order to use the only available graphics libraries for programing the GPU.

The purpose of this work is to employ the mobile GP-GPU for the development of an object detection library based on the Viola-Jones[5] approach. We make use of the many opportunities provided by the system-on-chip nature of a smartphone processing system.

II. BACKGROUND

Computer vision algorithms allow computers to understand their environment from visual inputs. The visual inputs can come from images, videos, or a camera. The purpose of computer vision is to interpret this visual input for some purpose. The main barrier to the advancement of computer vision is its requirement for large amounts of compute power. There is an ongoing effort to make the algorithms efficient and whenever possible utilize alternate computing facilities available in a platform. One such effort is the Open Computer Vision Collaboration (OpenCV)[6].

The Open Computer Vision collaboration is engaged in an on-going development of a set of libraries that can be used free of charge for many kinds of computer vision applications. The OpenCV library has more than 2500 computer vision algorithms incorporated. These algorithms are available for different processor architectures and different languages. It supports accelerating computer vision algorithms on GPUs using CUDA [7] and recently OpenCL support has been added (but only some algorithms are supported on mobile GP-GPUs). It also supports Windows, Linux, Mac OSX, iOS, and Android platforms.

Object detection algorithms are some of the algorithms found in the Open Computer Vision (OpenCV) library. The object detection algorithm based on the work by Viola and Jones [5] that uses Haar like features is the topic of this work. Listing 1 gives the algorithm for the object detection algorithm. The input to the algorithm is an equalized gray scale image.

Listing 1. Object Detection Algorithm [8]

1. Build integral image I(F) and square integral image SI(F)
2. Set curScale=1.0

978-1-5386-2775-4/17/$31.00 ©2017 IEEE

3. For (all scales)
    3.1. curScale*=S
    3.2. For (curRegion in all regions X on the current scale)
        3.2.1. For ($H_i(x)$ in all cascade stages)
            3.2.1.1. StageSum=0
            3.2.1.2. For ($h_j(x)$ in all weak classifiers of $H_i(x)$)
                3.2.1.2.1. StageSum+=calculate weak classifier $h_j(x$) using I(F) and IS(F)
            3.2.1.3. If(StageSum <StageThreshold)
                3.2.1.3.1. Mark region as non-object and proceed to next region
        3.2.2. Mark region as object
4. Partition and filter regions marked as objects

As can be seen in Listing 1, the object detection algorithm requires the computation of the integral image I(F) and square integral image SI(F) before the cascade classifier is applied. The integral image is computed as the cumulative sum of each pixel value and is stored in the respective pixel. A pixel in an integral image then contains all the intensity values that come before it and including it. Such approach reduces the computation of the sum of pixel values in a rectangular area just to four array references in the integral image. An integral image needs to be computed for all the scales of the original image. In addition, the original image needs to be resized for all the scales before computing the integral images. Of the two methods used in the OpenCV library, the bilinear [9] resizing method is used in this work.

III. Related Work

The literature shows that many researchers used the mobile GPUs for general-purpose applications. The graphics library OpenGL ES was the go to programming interface in the earlier days as the GPUs in the mobile devices did not support CUDA (on the NVidia Tegra platform) or OpenCL (on the other platforms) then. The literature in the following paragraphs clearly shows this trend..

Lee et. al [10] used mobile GPU for augmented reality application where they applied computer-vision techniques for tagging spaces for augmentation. Their work involved the learning of a patch of space to be augmented and then detecting and tracking the tagged space. In addition, the used the phones sensor's for pose estimation. They conclude that the GPUs in the phones enable a near real time Anywhere Augmentation.

Ensor and Hall [11] implemented the Canny edge detection on mobile GPUs using OpenGL ES 2.0. The implementation moved the entire pipeline in the Canny edge detector to the GPU. The Gaussian blurring, the gradient vector computation, the non-maximum suppression, double threshold and their own tweaks to the Canny algorithm are all done on the GPU. They report significant frame rate improvement with the GPU implementation for some of the mobile devices that were tested with 640x480 resolution.

Hofmann et. al [12] implemented an upright speeded up robust features (SURF) descriptor on a mobile GPU called uSURF-ES. They used OpenGL ES 2.0 and C++ for programming their application. Their implementation on the GPU was compared against the upright SURF in OpenCV, which is not multi-threaded. The comparison was made on different mobile devices and tablets, and speed-ups ranging from 2x upto 14x were reported.

The work by Cheng et. al [13] mapped the face recognition problem to a graphics-rendering paradigm. The face detection part of this work was done using an Android API. For the recognition part, they implemented the Gabor wavelet using Fast Fourier Transform. They used OpenGL ES on an NVidia Tegra SoC. The face recognition took about 8.5 seconds on the CPU while taking only 4.6 seconds on the GPU while consuming 16.3J of energy in contrast to the CPU only implementation's 29.8J. This shows an almost 2x speed up was gained using the GPU while also lowering the energy consumption by 45.3%.

The work by Rister et. al. [14] implemented the Scale-Invariant Feature Transform (SIFT) detector (which is often used in object detection [15]) on a mobile GP-GPU sing OpenGL ES. In their approach, data was partitioned between the CPU and GP-GPU in order to maximize efficiency. They used texture compression to reduce the data transfer requirement needed by their implementation by packing the grey image into an RBGA texture. They also report a significant energy consumption reduction (87%) as compared to CPU only implementation when using the heterogeneous combination. An implementation of the SIFT detector on mobile GPUs was also done by Wang et. al. [16]. They used OpenCL for their work instead of OpenGL ES. In their experiments they were able to achieve improved frame rates for key point detection and descriptor generations. About 41% energy consumption reduction was also reported when compared to an optimized C++ implementation on the CPU.

Object removal from images using an exemplar-based in-painting algorithm was implemented by Wang et. al. [17] on a mobile GP-GPU using OpenCL. They modified the object removal algorithm to be heterogeneous CPU-GPU application. Parts of the algorithm that take longer to compute were offloaded to the GPU. This heterogeneous implementation reduced the runtime from 393 seconds on the CPU coded in OpenCL to about 2 seconds on the GPU coded with OpenCL as well.

Jones et. al. [18] used mobile GPUs and OpenCL for acceleration of embodied robot simulation. They chose the Stage robot simulator's ray tracing algorithm to be accelerated using OpenCL as it was found to be the most time consuming. They report that 82% performance increase and around 30% drop in energy usage for one of their experiment setups. They also speculate that more performance and energy saving can be achieved with rigorous OpenCL code optimization as the goal of the current implementation was porting the ray tracing algorithm with minimal coding effort.

IV. Implementation on the GP GPU

The purpose of this work is to implement the object detection algorithm on a mobile GP-GPU and to measure the resulting performance improvement both in runtime and energy efficiency. The components of the algorithm described in Listing 1 are implemented on the GP-GPU as described in the subsequent sections.

A. Integral Image Computation

Before the integral image computation commences the original image needs to be resized to the appropriate scale. In order to do that the bilinear resizing method is used. The bilinear algorithm uses the average of the neighboring pixels to compute the value of the new pixel in the new resized image. This algorithm was straightforward to implement on the GPU. For the sake of efficiency, the original image was resized for all the scales beforehand and the resulting images were stored in memory.

Once the image is resized, the next step is to compute the integral images. In order to compute the integral image on the GP-GPU, the prefix sum [19] method is used. The prefix method uses a down ward reduction operation on a summation tree and an upward summation pass. The computation of the integral image on the GPU in this way requires the computation of the prefix sum on the rows of the image first and then followed by the columns. However, doing this directly will be very inefficient, as data locality will be affected during the computation of the column prefix sum. Transposing the image after the computation of the row prefix sum then solves the problem of data locality. Another transpose was done to return the integral image to an upright position after the computation of the column prefix sum. In this work, the row prefix sum and the first transpose as well as the column prefix sum and the second transpose were combined in a single kernel each to reduce the cost of kernel call overhead. This means we will have only two kernel calls instead of four.

The threads on the GP-GPU for the computation of row and column prefix sums were organized as [image height x 8] and a workgroup of 8x8 work-items were used. Each row of eight threads loops over a row of the image computing the prefix sums. This way the need for synchronization and further need for computation to generate a complete prefix sum over the row is avoided as compared to if the threads were organized as [image height x image width].

B. Searching for Objects

The next step in the object detection algorithm is the search for objects in given image. As already discussed, the image has been resized and integral image computed for different scales and is stored in memory beforehand. Doing so reduces the need to allocate memory, resize the image, and compute the integral image on the fly for each scale, although this has a toll on memory requirement.

A similar thread organization as in the case of the integral image computation is used when searching for the objects. In order to search for the object, each thread will apply the cascade classifier to an assigned region of the image called a window for each scale. In the naïve implementation of this approach each thread applied the entire cascade classifier on each window that results in some threads finishing work early and waiting on all other threads that need to do further processing.

As shown in [20], the first stage of the classifier is processed on all the windows by all the threads. But not many of the windows actually 'pass' the first stage test (meaning the object is not likely to be found in this window and further processing is not required). As a result, the algorithm shown in Listing 1 was modified in such a way that all threads test the first stage on all windows and then put the windows that pass the first stage into a work queue. Once the processing of all windows for the first stage is done, the same threads can be utilized to pick work from the work queue to test for the second stage and put back those that pass the second stage on the work queue, so on. With this approach, the number of threads that are stalled waiting on other threads to finish is significantly reduced. We have two implementations of this approach on the GPU. The first version directly access the integral image stored in the global memory of the GPU and does the computation there. The second version first copies blocks of the image in global memory to the local memory of the GPU for use in computation.

V. Methodology and Experiments

The Qualcomm Snapdragon 805 [21] mobile development tablet from Intrinsyc was used in this work. It has four Krait 450 processors that run at speeds of up to 2.5 GHz, with 3GB of LPDDR3 memory running at 800MHz. It also contains the Adreno 420 GPU running at 600MHz and this GPU boasts 32KB local memory per compute unit. Moreover, both the processor and the GPU share the same RAM. This is important as it allows us to avoid memory copies as we can access the same memory from the CPU and GPU.

The OpenCV library that we compared to in this work was version 3.1. It has been compiled with multi-threading and OpenCL enabled for Android. But while we have confirmed the multi-threading works well, we were unable to make the OpenCL version of the object detection from OpenCV work on the mobile device. The OpenCL object detection version was found to work on the desktop environment when the same version of OpenCV was compiled for the desktop environment with the same settings as in the case of the Android version.

For accurate timing measurements the software has been run 50 times for each test image and the average has been taken. In order to measure the energy consumption, we have used a data logger from National Instruments to measure the current drawn by the tablet when the object detection application was running. The screen of the tablet was kept turned of at all times and the tablet has been left idle for sometime to measure the current drawn at rest.

The first experiment conducted was to measure the runtime performance of the different implementations of the object detection algorithm. We have tested the OpenCV Object Detection that is multithreaded on the CPU (listed as OpenCV-CPU in the results section below), our serial implementation on

	OpenCV CPU		MyCPU			MyGPU RTC			MyGPU RTC LM		
Images Res - #Object	Runtime (S)	Detections	Runtime (S)	Detections	Speed Up	Runtime (S)	Detections	Speed Up	Runtime (S)	Detections	Speed Up
Full HD - 1	0.63	1	3.58	1	0.18	0.26	1	2.39	1.07	1	0.59
Full HD - 2	1.29	2	6.85	3	0.19	1.42	1	0.91	1.39	1	0.93
Full HD - 3	1.29	3	6.12	3	0.21	0.59	3	2.20	1.33	2	0.97
Full HD - 9	1.30	9	7.26	9	0.18	1.02	7	1.28	1.56	6	0.83
Full HD - 19	1.35	19	7.50	20	0.18	1.25	16	1.09	1.62	13	0.83
Full HD - 72	1.83	72	8.24	72	0.22	1.75	65	1.05	1.63	111	1.12
512x512 - 1	0.16	1	0.75	1	0.22	0.20	1	0.81	0.20	1	0.80
450x326 - 2	0.08	2	0.41	2	0.19	0.13	1	0.61	0.18	1	0.44
647x650 - 31	0.30	31	1.59	31	0.19	0.48	21	0.63	0.46	33	0.66

Table 1 – Runtime Performance Measurement

the CPU (MyCPU), our implementation on the GPU that has reduced thread count with work reduction (MyGPU RTC – as described in section IV) and a reduced thread count with work reduction on the GPU with local memory (MyGPU RTC - LM). These algorithms were tested with images of different sizes ranging from small to Full HD resolution. In addition the images had a range of different numbers of objects being searched for. The objects of interest in these experiments were human faces. We have used the HAAR based classifier for faces that comes with OpenCV. There is one difference between our algorithm and the general OpenCV one:- our work is limited to non-tilted features, meaning we can not find faces that are at an angle.

We measured the energy consumption of the different algorithms in the following way: the current drawn by the tablet was measured while the implementations listed were being run on the different images. In order to have reliable measurements, each implementation was run for multiple times as in the case of the runtime measurement. However, the number of runs was dependent on the runtimes associated with the test images. We used 20 runs for images that took longer to process and 50 runs for images with lower runtimes. This was done to have reasonable current measurement samples from the data logger. Then the root mean square power usage was computed from the collected data to arrive at the energy consumption values.

VI. Results

A. Runtime Performance

In this section, we present the results obtained from the runtime performance and energy performance measurements. The runtime measured, the number of detections and the speed up against the baseline OpenCV CPU are given in Table 1. The runtime measurements include the time it takes to compute the resizing, the integral images and searching for objects in the image for all scales of the image. We have observed the standard deviation of multiple runs to be with in 9% of the average runtimes overall and less than 1% for MyGPU RTC implementation. Another observation is that the serial implementation of the object detection algorithm, (called MyCPU) is slower compared to the baseline OpenCV implementation by almost 5 times. Note also that it detects the same number of faces as the OpenCV version mostly.

The MyGPU RTC implementation has speed ups of up to more than two times for Full HD resolution images. However, for images that are smaller it is actually slower because overhead of launching the object detection on the GPU for smaller images out weighs the benefit. Similarly as the number of objects in test images of Full HD resolution increases, the performance of the MyGPU RTC shows a decline in performance. This is caused by the fact that more and more candidate windows pass for further processing in the classifier cascade. One solution for this might be a heterogeneous computing with GPU and CPU where the candidate windows that pass the test on latter stages of the cascade classifier can be pushed to the CPU for processing. Another observation for this implementation is that there are fewer number of detections compared to the OpenCV CPU implementation. Such behavior is caused by the limited size of local memory that is necessary to store the work queue.

The MyGPU RTC LM implementation, which tried to utilize the high performance local memory that exists in the Adreno GPU used in this experiments, was found to perform poorly with regards to runtime performance. The cause for this performance loss is the nature of the object detection algorithm used. This algorithm uses the sliding windows approach which means one has to copy adjacent windows to the local memory as illustrated by the following example. Since the work-items are grouped in a 8x8 workgroup size, one has to copy 8x8 = 64 adjacent windows to the local memory from the GPU's global memory. Each window is in turn 20x20 pixels which means a 28x28 block of image has to be copied to the local memory for every set of workgroup. This leads to a huge amount (about 92%) of redundant memory copies that lead to significant performance loss. In our implementation we have tried to reduce this overhead by making copies of sixteen 8x8 blocks of windows per workgroup in MyGPU RTC LM. Even though the redundant memory copies were reduced to 62% with this approach, performance was still reduced as the 62% redundant copy is still significant.

	OpenCV CPU	My CPU		MyGPU RTC		MyGPU RTC LM	
Images	**Energy(J)**	**Energy(J)**	**% Improvement**	**Energy(J)**	**% Improvement**	**Energy(J)**	**% Improvement**
Full HD Object - 1	3.81	8.42	-121.00	0.61	83.99	2.23	41.47
Full HD Object - 2	7.66	16.15	-110.84	3.38	55.87	2.85	62.79
Full HD Object - 3	7.71	14.30	-85.47	1.33	82.75	2.65	65.63
Full HD Object - 9	7.89	17.29	-119.14	2.41	69.46	3.24	58.94
Full HD Object - 19	8.08	17.62	-118.07	2.88	64.36	3.35	58.54
Full HD Object - 72	9.78	19.56	-100.00	4.07	58.38	3.92	59.92
512x512 - 1	0.80	1.76	-120.00	0.44	45.00	0.42	47.50
450x326 - 2	0.43	0.95	-120.93	0.28	34.88	0.38	11.63
647x650 - 31	1.68	3.68	-119.05	1.07	36.31	0.92	45.24

Table 2 – Energy Performance Measurement

B. Energy Efficiency Measurement

In this measurement, the energy consumed by the tablet while running the different implementations on the test images used in this work is provided. Table 2 gives the energy consumption measured in Joules and the improvement in percentage as compared to the baseline, which is the OpenCV CPU implementation. An immediate observation from the data in Table 2 is that serial or multi-threaded implementations on the CPU consistently consumed more energy compared to the implementations on the GPU. The serial implementation MyCPU consumed almost only twice as much as the multithreaded OpenCV CPU implementation. This is a surprise because OpenCV CPU was almost 5x faster than MyCPU. The reason for this behavior may arise from the fact that all four cores of the CPU are activated and the device draws more current from the supply for the OpenCV version. It was observed that the multi-threaded OpenCV CPU drew around 500mA while MyCPU drew about 200mA.

Both the GPU based implementations consistently showed energy efficiency improvement over the OpenCV CPU version. In particular, the best energy efficiency (about 84%) improvement was achieved for MyGPU RTC when run with a Full HD image with only one object in the scene. This can be attributed to the nature of the algorithm that rejected most of the candidate windows at the early stages of the cascade classifier. Lower energy efficiency improvements were measured for the smallest resolution images although they are still significantly better than the OpenCV CPU implementation.

The MyGPU RTC LM had a maximum of 66% and a minimum of 11% energy consumption improvement over the baseline. Recall from the previous section that MyGPU RTC LM was mostly slower in runtime compared to the OpenCV CPU implementation. This shows that even when an implementation is underperforming in runtime on the mobile GP-GPU, there is a higher chance that energy can be saved by pushing some computation to the GPU. This suggests that for non-real time applications using the mobile GP-GPU will definitely be beneficial in conserving battery charge levels.

VII. CONCLUSION

We have shown performance measurements of the Viola-Jones-based object detection on mobile GP-GPUs. It is shown that one can achieve the up to more than twofold speedup while improving energy efficiency by up to 84% when offloading general-purpose application to the mobile GP-GPU. These results were found out to be in line with what is found in the literature as well. In the future, we plan to apply the object detector developed in this work for applications that are non-real time in the area of health and agriculture.

REFERENCES

[1] Qualcomm Inc., Snapdragon 805 Processor. https://www.qualcomm.com/products/snapdragon/processors/805 (accessed May 08, 2017)

[2] Mali graphics processing for ARM. http://www.arm.com/products/graphics-and-multimedia/mali-gpu (accessed May 08, 2017)

[3] PowerVR Graphics. https://www.imgtec.com/powervr/graphics/ (accessed May 08, 2017)

[4] Khronos OpenCL Working Group, “The OpenCL Specification”, Version 2.0, 2014

[5] Viola, Paul, and Michael Jones. "Rapid object detection using a boosted cascade of simple features." Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on. Vol. 1. IEEE, 2001.

[6] Open Computer Vision. http:/opencv.org (accessed May 08, 2017)

[7] Nvidia, C. U. D. A. "Programming guide." (2008).

[8] Wen-Mei, W. Hwu. *GPU computing gems emerald edition*. Elsevier, 2011.

[9] Lehmann, Thomas Martin, Claudia Gonner, and Klaus Spitzer. "Survey: Interpolation methods in medical image processing." *IEEE transactions on medical imaging* 18.11 (1999): 1049-1075.

[10] Lee, Wonwoo, et al. "Point-and-shoot for ubiquitous tagging on mobile phones." *Mixed and Augmented Reality (ISMAR), 2010 9th IEEE International Symposium on*. IEEE, 2010.

[11] Ensor, Andrew, and Seth Hall. "GPU-based image analysis on mobile devices." *arXiv preprint arXiv:1112.3110* (2011).

[12] Hofmann, Robert, Hartmut Seichter, and Gerhard Reitmayr. "A GPGPU accelerated descriptor for mobile devices." *Mixed and Augmented Reality (ISMAR), 2012 IEEE International Symposium on*. IEEE, 2012.

[13] Cheng, Kwang-Ting, and Yi-Chu Wang. "Using mobile GPU for general-purpose computing–a case study of face recognition on smartphones." In VLSI Design, Automation and Test (VLSI-DAT), 2011 International Symposium on, pp. 1-4. IEEE, 2011.

[14] Rister, Blaine, Guohui Wang, Michael Wu, and Joseph R. Cavallaro. "A fast and efficient SIFT detector using the mobile GPU." In Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on, pp. 2674-2678. IEEE, 2013.

[15] Lowe, David G. "Object recognition from local scale-invariant features." InComputer vision, 1999. The proceedings of the seventh IEEE international conference on, vol. 2, pp. 1150-1157. Ieee, 1999.

[16] Wang, Guohui, Blaine Rister, and Joseph R. Cavallaro. "Workload analysis and efficient OpenCL-based implementation of SIFT algorithm on a smartphone." InProceedings in IEEE global conference signal and information processing (GlobalSIP), pp. 759-762. 2013.

[17] Wang, Guohui, Yingen Xiong, Jay Yun, and Joseph R. Cavallaro. "Accelerating computer vision algorithms using OpenCL framework on the mobile GPU-a case study." In Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on, pp. 2629-2633. IEEE, 2013.

[18] Jones, Simon, Matthew Studley, and Alan Winfield. "Mobile GPGPU acceleration of embodied robot simulation." Artificial Life and Intelligent Agents Symposium. Springer International Publishing, 2014.

[19] Harris, Mark, Shubhabrata Sengupta, and John D. Owens. "Parallel prefix sum (scan) with CUDA." *GPU gems* 3.39 (2007): 851-876.

[20] Micikevicius, Paulius, "Maximizing Face Detection Performance", GPU Technology Confernece, 2015

[21] MDP Tablet based on the Qualcomm® Snapdragon™ 805 Processor by Qualcomm Technologies, Inc., http://www.intrinsyc.com/snapdragon-development-platforms/mdp-805-tablet/,(accessed April 19, 2014)