Performance and energy characterization of high-performance low-cost cornerness detection on GPUs and multicores

Apostolos Glenis
Institute of Computer Science
FORTH
Heraclion, Greece
Email: aglenis@ics.forth.gr

Sergios Petridis
Institute of Informatics and Telecommunications
NCSR "Demokritos"
Agia Paraskevi, Greece
Email: petridis@iit.demokritos.gr

***Abstract*—Feature detection and tracking is an important problem in Computer Vision. Corners in an image are a good indication of features to track. Original algorithms may be expensive even on multicore architectures because they require full convolutions to be performed. Although these can be performed in real time in modern GPUs and multicore CPUs, faster solutions are needed for embedded systems and complex algorithms, given that corner detections is just a step of the analysis process. In this paper we evaluate the performance and energy efficiency of the Harris corner detection algorithm as well as an approximation of it, in both desktop and mobile platforms. The purpose of this paper is three-fold: evaluate the performance gains of GPUs vs. CPUs for several mobile and desktop systems, evaluate whether the Harris approximation provides adequate performance gains to justify its use in mobile and desktop system configurations and, finally, determine which configurations provide real-time performance. According to our evaluation (a) the best GPU solution is 16.3 times faster than the best CPU solution for the desktop case while being 2.6 times more energy efficient and (b) the best GPU solution for the mobile case is 1.2 times faster while being 3.6 times more energy efficient than the respective CPU.**

***Keywords*-Harris corner detection, CUDA, mobile computing**

I. Introduction

From the time programmable GPUs emerged to the market, several successful attempts have been made in porting Computer Vision algorithms to them and exploring the performance and energy gains from their use [1]–[6]. As mobile platforms become more pervasive, computer vision algorithms get ported to mobile platforms that use a GPU to exploit the available data parallelism [7]–[13]

Feature detection is an important but computationally intensive step of computer vision algorithms. This leads to the need for approximation algorithms especially for embedded platforms. As Moore's law fails to scale, chip manufacturers cramp more cores into a single Central Processing Unit (CPU) chip to provide scalability. However, as more chips gets cramped into a single core, energy efficiency becomes a major concern. All the above have lead to algorithms being ported to Graphical Processing Units (GPUs) because of their design. The Harris feature detection algorithm [14] has massive amounts of parallelism and therefore is a great candidate for porting to a massively parallel architecture such as modern GPUs. In this paper we evaluated both the initial Harris corner detection algorithm as well as a fast variant [15] in terms of their performance and energy efficiency in desktop and mobile platforms equipped with a variety of CPUs and GPUs. The paper is structured as follows: Section II describes the implementation of the Harris detector versions used for comparison, including details on GPU optimizations. Section III presents the results of our performance evaluation and, finally, in Section IV we describe ways in which the detector can be improved as well as future research directions.

II. Feature detection using the Harris Kernel

A. The baseline algorithm

As a reference implementation, Harris feature detector has been considered as implemented in the OpenCV Computer Vision library [16], though with some modifications, so that a comparison with the fast alternative would be meaningful. In partucular, to provide a fair comparison, the OpenCV Harris feature detector has been reimplemented. Hereafter, we will referred to this implementation as the "baseline" Harris implementation. The Harris feature detection, as we implemented it, is comprised of three steps:

1) Sobel edge detection to approximate the gradients in the x and y direction
2) Application of the Harris cornerness detection kernel (with a 9×9 window for our experiments)
3) Non-maxima suppression to avoid corners with extreme spatial locality.

B. The fast approximation

The main advantage of the approximate Harris Detector is it's reduced complexity compared to the baseline algorithm. Although Gaussian convolution can be performed as two separate filters, thus reducing the complexity substantially, using integral images [17] we can compute an approximation

Table I
REDUCTION OF THE COMPUTATION COST FOR THE CONVOLUTION

	Additions	Mult/ions	Total
Gaussian	$2 \cdot N$	$2 \cdot N$	$4 \cdot N$
LC-Harris	$2 + 3 \cdot 2$	1	9

Table II
REDUCTION OF THE COMPUTATION COST FOR THE CORNERNESS DETECTION

	Additions	Mult/ions	Total
baseline	$3 \cdot w \cdot w$	$3 \cdot w \cdot w$	$6 \cdot w \cdot w$
LC-Harris	$3 \cdot (2 + 3) + 3$	$3 + 4$	$18 + 7$

Table III
TWO-DIMENSIONAL PREFIX-SCAN ON THE GPU (MILLISECONDS)

Resolution	CUB	CUDPP	THRUST
1024×1024	0.12	0.16	1.08
4096×4096	1.68	2.23	10.5

Table IV
INTEGRAL IMAGE COMPUTATION ON THE GPU (MILLISECONDS)

Resolution	CUB	CUDPP	THRUST
1024×1024	0.4	0.35	2.03
4096×4096	5.7	6.7	23

of the Gaussian derivative with constant number of operations per pixel at any window size. This also applies to the cornerness detection mechanism, where using integral images reduces complexity with no actual cost in accuracy. Using box filters, we can have great benefits for embedded platforms where computational resources are limited.

The approximation algorithm is based on the notion that using integral images we can sum an area of an image in constant time. The steps of the algorithm are as follows:

1) Calculate the integral image of the original image.
2) Compute the gradients gx and gy using the integral image for the Gaussian kernel approximation.
3) Compute the integral image of gx2 , gy2 and gxgy to use for the cornerness metric evaluation.
4) Evaluate the cornerness response R for each pixel of the image.
5) Perform non-maxima suppression to obtain the final cornerness image

The main difference between the two algorithms is that the approximation algorithm uses integral images to compute the gradients needed to evaluate the cornerness response. This means that we end up doing three convolutions less than the original algorithm. The added speedup comes with a space overhead since now we need to store four additional arrays for the integral images. With the approximate algorithm the window size does not affect complexity.

C. *Optimizations in the GPU version*

The algorithm in question has been ported to GPUs by [18]. Our initial implementation closely resembles the one presented in that paper.

A key part in the algorithm we consider is computing integral images. There are a few ways of performing integral image computations. The most common and easy to implement is to do a prefix scan followed by a matrix transpose, followed by a prefix scan, and then another transpose to fix the orientation of the image as proposed by [19]. Another more efficient approach that uses tiling is proposed in [20].

We tried to optimize the baseline GPU implementation by using different available libraries to compute integral images. Our original CUDA implementation uses CUDPP [21] for a high performance two-dimensional prefix scan implementation. In addition to the CUDPP implementation of integral images the programmer can use either CUB [22] or Thrust [23] to implement a two dimensional prefix scan operation.

To decide which implementation to use, we evaluated our optimizations to the GPU version on the desktop with GTX480, to choose the best way of performing the integral image calculation (see Section III-A for details about the system descriptions). The CUDPP implementation tested is similar to the one proposed by [19]. Our results, shown at Table III, indicate that Thrust's approach is not optimal and CUDPP and CUB perform almost identically. As Table IV shows the results are similar to the prefix-scan case but the effect of the prefix scan is mitigated by the cost of the transpose steps of the algorithm.

In conclusion, the CUB implementation gave the best performance so we used that for the evaluation.

III. EVALUATION

A. *Test cases*

The primary system for testing was employed with a GeForce GTX480 with 1.5 GB of RAM and a GTS450 with 1 GB of RAM. The CPU was an Intel Core2Duo operating at 3.6Ghz with 4GB of RAM. A secondary system with an Intel i7 was used to evaluate the algorithm on a more modern system, and see how a faster system would affect the algorithms in question. For the mobile versions we used two systems:

1) a ZOTAC ZBOX ID84 with a CedarTrail Atom chipset, a dual-core D2550 1.86 GHz processor and Nvidia Geforce GT 520M
2) an ASUS U36JC with Intel Core i5 480M operating at 2.66 GHz and a Nvidia 310m

A summary of the different processors used for evaluation is shown at Table V

B. *Evaluation Metrics*

We used two metrics to perform out evaluation : Frames per Second and Performance per Watt.

Table V
CHARACTERISTICS OF THE PROCESSORS USED FOR EVALUATION

Architecture	Processing Cores	Clock Frequency (Ghz)	Max memory bandwidth (GB/s)	TDP(W)
Atom	2	1.86	6.4	10
GTX480	448	1.215	133	250
GTS450	192	1.566	57	106
core2duo	2	3.6	7	65
310m	16	1.53	9.1	14
520m	48	1.6	14.4	12
i7	8	3.5	25.7	77
i5	4	2.66	17.1	35

Table VI
SERIAL VS GPU IMPLEMENTATION

resolution	speedup compute core2duo	speedup compute i7	speedup mem core2duo	speedup mem i7
512×512	16.3	7.8	12.8	6.1
1024×768	16.2	11.8	12.6	9.2

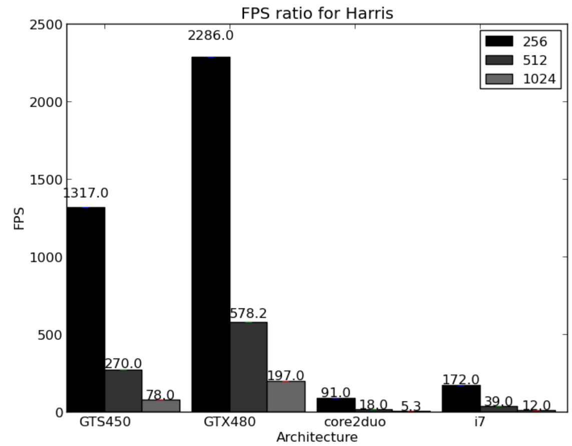
- **Frames per Second** (FPS) is the total number of video frames processed divided by the total (wall) time taken to process them.
- **Performance per Watt** is defined as the FPS divided by the Thermal Design Power (TDP) of the processor.

C. Results for the desktop-case experiments

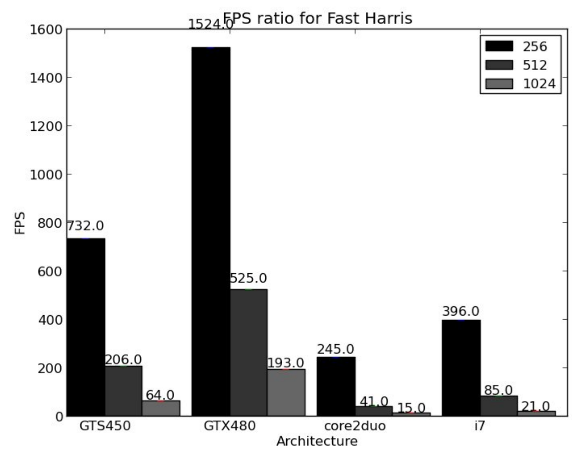
1) Execution Times:

CPU vs GPU: The GPU implementation provides significant speedup compared to the serial implementation in both test systems. The data transfer overhead seems to matter a lot less as the image size increases, which is evident because speedup for the computation is getting closer to the speedup measured when including the data transfer. This happens because as the resolution increases, the computation part of the algorithm becomes a lot more time consuming than the data transfer to and from the GPU. Table VI shows that our detector is 16 times faster than its serial counterpart in the core2 system and 12 times faster in the i7 system. As shown in Figures 1(a) and 1(b), the GPUs are consistently much faster than the CPUs. Also the high-end GPU performs roughly two times better than the mid-end GPU, which was expected according to their specifications given that GTX480 has twice the amount of processing cores and memory bandwidth than GTS480.

Baseline versus fast implementation on the CPU: The approximate algorithm is twice as fast as the baseline Harris detector and the difference would have been even greater if we had used larger window sizes. Comparing the two different processor we observe that the increased memory bandwidth in the i7 makes the algorithm twice as fast.



(a) Fast Harris



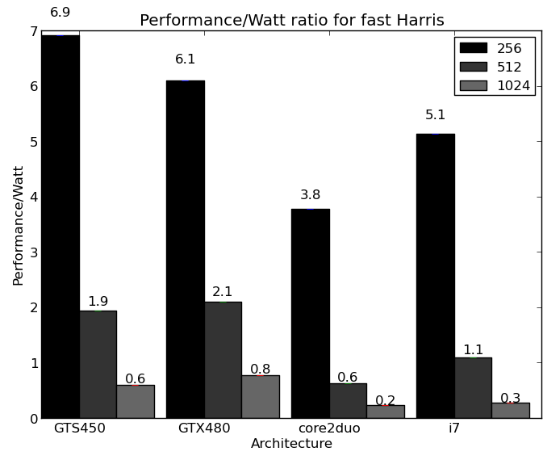
(b) Baseline Harris

Figure 1. FPS results in desktop system

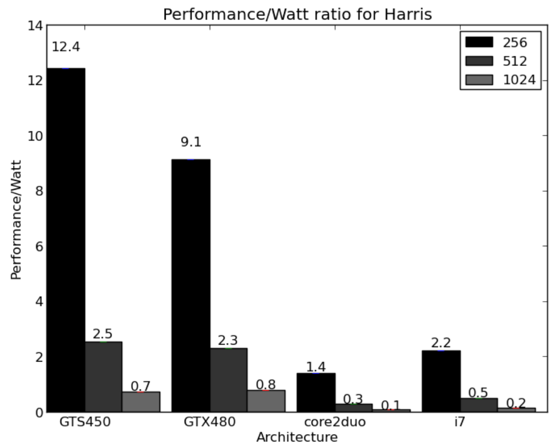
Table VII
REALTIME CONFIGURATIONS FOR THE DESKTOP SYSTEM

	Baseline Harris		Fast Harris	
Architecture	Resolution	fps	Resolution	fps
GTX480	1024×1024	197	1024×1024	193
GTS450	1024×1024	78	1024×1024	64
core2duo	256×256	91	512×512	41
i7	512×512	39	512×512	85

Baseline versus fast implementation on the GPU: The fast corner detector performs worse than the baseline on desktop GPUs. We speculate that this happens for a number of reasons:the window size used for edge detection as well as the cornerness evaluation in the baseline Harris is relatively small so even the theoretical gains are slim for the edge detection phase of the algorithm. Also the baseline Harris requires less steps and less bookkeeping to complete so if the computational complexity of the convolution is hidden



(a) Fast Harris



(b) Baseline Harris

Figure 2. Energy Efficiency for the Desktop System

because of the parallelism then the baseline Harris can achieve better performance than the approximate algorithm.

2) Energy Efficiency: After comparing the energy efficiency of the two GPUs we noticed that GTX480 has almost double the performance in all configurations compared to the GTS450. In the small resolution the difference is slightly lower because the high-end GPU is not fully occupied. In the first resolution the GTS450 is visibly better in the performance per watt comparison as the resolution increases GTX480 recovers the lost ground and finally wins in the larger resolution. From the two remarks above we can conclude the high-end GPU is not always the best choice when energy costs are taken into account as show in Figures 2(a) and 2(b). Moving to the CPU results the i7 processor is consistently two times faster both in Harris and fast Harris compared to the core2duo and because they consume roughly the same power the i7 is roughly two times more energy efficient as shown in Figures 2(a) and 2(b).

Table VIII
REALTIME CONFIGURATIONS FOR THE MOBILE SYSTEMS

Architecture	Baseline Harris Resolution	fps	Fast Harris Resolution	fps
Atom	N/A		256×256	38
i5	256×256	89	512×512	49
310m	512×512	25	256×256	89
520m	512×512	67	512×512	40

3) Conclusions: In baseline Harris we have to limit the resolution to 512×512 for the i7 to sustain real-time performance and to 256×256 for the core2duo to remain real-time (512×512 is 17fps) The GPUs are well above realtime for all configurations. We should also note that the GPUs are much less sensible to the increased algorithmic complexity of the baseline Harris. A detailed view of the real-time configurations is presented in table VII.

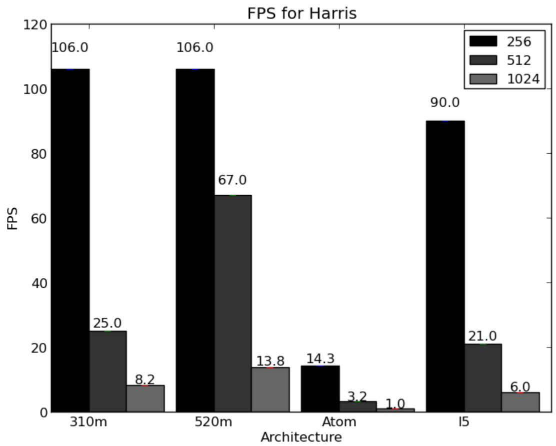
D. Results for the mobile case experiments

1) Execution Time: The i5 ULV processor performs better than both GPU solutions in the two small resolutions for Fast Harris, but the 520m performs better in the larger resolution.The difference between the ULV processor and 310m get smaller as the resolution increases. This is probably because the GPUs get utilized better, as more parallelism is exposed from the application. In the baseline Harris version, the ULV still outperforms the GPUs in the small resolution but loses the battle from both GPUs as the resolution increases. This happens because of the increased algorithmic complexity hidden in the GPU. With fast Harris i5 is realtime up to 512×512 and with baseline Harris is real time at 256×256.The 310m is realtime at 512×512. Atom is real-time only at 256×256 even with the use of the approximate algorithm. Finally the 520m is also real-time at 512×512 but with 40fps. When running the baseline Harris the CPUs are having a hard time staying above the real-time baseline. The ULV is real time only at 256×256. Atom is not real-time at any resolution making the use of the approximate algorithm mandatory for real-time applications. In baseline Harris, both GPUs are real-time at 512×512 and faster than the approximate algorithm for the reasons stated in the introduction of the paper. A detailed view of the real-time configuration for the mobile systems is presented in Table VIII.

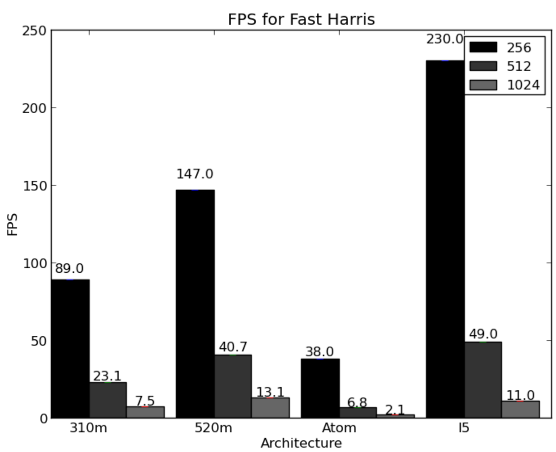
2) Energy efficiency: Both GPUs are far more energy efficient that the ULV processor and Atom both in fast and baseline Harris as shown in Figures 4(a) and 4(b). Atom has the worst performance in the energy efficiency proving that it is a processor targeted at providing low power consumption, not performance under an energy budget.

E. Conclusions

As stated above GPUs are of great value to mobile world because they allow us to perform complex algorithms in re-

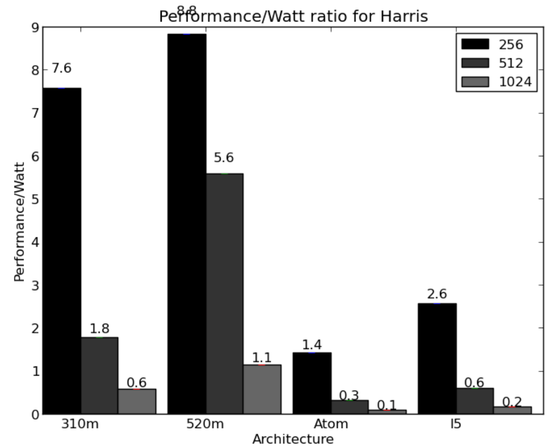


(a) Baseline Harris

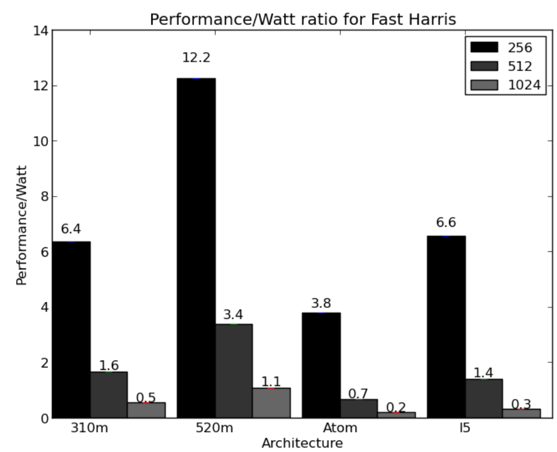


(b) Fast Harris

Figure 3. FPS results for the Mobile System



(a) Baseline Harris



(b) Fast Harris

Figure 4. Energy efficiency for results for the Mobile System

altime for resolutions that it wouldn't be possible otherwise. Also they are very good at providing performance with an energy budget. It is worthwhile to mention that while the mobile GPUs are as energy efficient as desktop GPUs, if not less, they provide performance under a far more constrained energy envelope.

IV. Conclusions

In this paper, we compared the merits of a high performance and low complexity corner feature detector against a number of different desktop and mobile platforms. We implemented the detector for GPUs as well as CPUs and we evaluated it's performance against the baseline not approximate version of the Harris Feature Detector. Furthermore we evaluated several optimizations for parts of the algorithm on the GPU. Finally we evaluated the energy efficiency of each processor architecture for Harris feature extraction. According to our evaluation Harris feature extraction is a good match for GPUs both in terms of performance and energy efficiency. Mobile processors like Atom are unsuited for heavy computation but ULV processors strike a good balance between performance and energy efficiency and remain real-time for relatively small resolutions. According to our evaluation mobile platforms can benefit greatly from adding a GPU for data-parallel workloads like Harris Corner Detection. Comparing the approximate and the baseline Harris detector we conclude that the fast detector is a perfect match for CPU architecture but the baseline Harris can perform better than fast Harris for small window sizes because there is no need for auxiliary operation and the extra computation cost is masked by the parallelism available.

As future work, we plan on evaluating the potential gains of overlapping computation and data transfer as well as multi-GPU support and also compare the qualitative differences between the baseline algorithm and the approximation. During our tests both the 'fast' and the 'slow' version of the detector yielded similar features in our sample video. In our opinion there is no significant reason not to use the 'fast version' of the detector. This empirically validates the claims of the original paper for the fast detector. As future work we plan to quantify the two solutions in terms of repeatability and other metrics, similar to [24].

Acknowledgment

This work was partially supported by the EU FP7-ICT-2011-9-601165 project WEARHAP.

References

[1] R. Kalarot and J. Morris, "Comparison of fpga and gpu implementations of real-time stereo vision," in *Computer Vision and Pattern Recognition Workshops (CVPRW), 2010 IEEE Computer Society Conference on*. IEEE, 2010, pp. 9–15.

[2] J.-S. Kim, M. Hwangbo, and T. Kanade, "Realtime affine-photometric klt feature tracker on gpu in cuda framework," in *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*. IEEE, 2009, pp. 886–893.

[3] M. Schweitzer and H.-J. Wuensche, "Efficient keypoint matching for robot vision using gpus," in *Computer Vision Workshops (ICCV Workshops), 2009 IEEE 12th International Conference on*. IEEE, 2009, pp. 808–815.

[4] K. Ogawa, Y. Ito, and K. Nakano, "Efficient canny edge detection using a gpu," in *Networking and Computing (ICNC), 2010 First International Conference on*. IEEE, 2010, pp. 279–280.

[5] L. Mussi, F. Daolio, and S. Cagnoni, "Evaluation of parallel particle swarm optimization algorithms within the cuda architecture," *Information Sciences*, vol. 181, no. 20, pp. 4642–4657, 2011.

[6] O. Kalentev, A. Rai, S. Kemnitz, and R. Schneider, "Connected component labeling on a 2d grid using cuda," *Journal of Parallel and Distributed Computing*, vol. 71, no. 4, pp. 615–620, 2011.

[7] K. Pulli, A. Baksheev, K. Kornyakov, and V. Eruhimov, "Real-time computer vision with opencv," *Communications of the ACM*, vol. 55, no. 6, pp. 61–69, 2012.

[8] K.-T. Cheng and Y.-C. 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*. IEEE, 2011, pp. 1–4.

[9] N. Singhal, I. K. Park, and S. Cho, "Implementation and optimization of image processing algorithms on handheld gpu," in *Image Processing (ICIP), 2010 17th IEEE International Conference on*. IEEE, 2010, pp. 4481–4484.

[10] Y.-C. Wang, B. Donyanavard, and K.-T. T. Cheng, "Energy-aware real-time face recognition system on mobile cpu-gpu platform," *Trends and Topics in Computer Vision*, pp. 411–422, 2012.

[11] C.-H. Chou, P. Liu, T. Wu, Y. Chien, and Y. Zhao, "Implementation of parallel computing fast algorithm on mobile gpu," *Unifying Electrical Engineering and Electronics Engineering*, pp. 1275–1281, 2014.

[12] U. Khan, M. Quaritsch, and B. Rinner, "Design of a heterogeneous, energy-aware, stereo-vision based sensing platform for traffic surveillance," in *Intelligent Solutions in Embedded Systems (WISES), 2011 Proceedings of the Ninth Workshop on*. IEEE, 2011, pp. 47–52.

[13] B. Rister, G. Wang, M. Wu, and J. R. Cavallaro, "A fast and efficient sift detector using the mobile gpu," in *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE, 2013, pp. 2674–2678.

[14] C. Harris and M. Stephens, "A combined corner and edge detector." in *Alvey vision conference*, vol. 15. Manchester, UK, 1988, p. 50.

[15] P. Mainali, Q. Yang, G. Lafruit, R. Lauwereins, and L. Gool, "Lococo: Low complexity corner detector," in *Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on*. IEEE, 2010, pp. 810–813.

[16] G. Bradski and A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc., 2008.

[17] F. C. Crow, "Summed-area tables for texture mapping," in *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3. ACM, 1984, pp. 207–212.

[18] P. Rondao Alface, "Low complexity corner detector using cuda for multimedia applications," in *MMEDIA 2011, The Third International Conferences on Advances in Multimedia*, 2011, pp. 7–11.

[19] B. Bilgic, B. K. Horn, and I. Masaki, "Efficient integral image computation on the gpu," in *Intelligent Vehicles Symposium (IV), 2010 IEEE*. IEEE, 2010, pp. 528–533.

[20] D. Nehab, A. Maximo, R. S. Lima, and H. Hoppe, "Gpu-efficient recursive filtering and summed-area tables," *ACM Transactions on Graphics (TOG)*, vol. 30, no. 6, p. 176, 2011.

[21] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson, "Cudpp: Cuda data parallel primitives library," 2007.

[22] D. Merrill. Cub cuda libary. [Online]. Available: http://nvlabs.github.io/cub/

[23] N. Bell and J. Hoberock, "Thrust: A productivity-oriented library for cuda," *GPU Computing Gems*, vol. 7, 2011.

[24] C. Schmid, R. Mohr, and C. Bauckhage, "Evaluation of interest point detectors," *International Journal of computer vision*, vol. 37, no. 2, pp. 151–172, 2000.