Low Complexity Corner Detector Using CUDA for Multimedia Applications

Rajat Phull, Pradip Mainali, Qiong Yang
Interuniversitair Micro-Electronica Centrum vzw.
Interdisciplinary Institute for BroadBand Technology
Kapeldreef 75, Leuven B-3001, Belgium
rajatphull@gmail.com, {pradip.mainali,qiong.yang}@imec.be

Patrice Rondao Alface
Alcatel-Lucent Bell Labs
Copernicuslaan 50, Antwerp B-2018, Belgium
patrice.rondao_alface@alcatel-lucent.com

Henk Sips
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
sips@ewi.tudelft.nl

Abstract—**High speed feature detection is a requirement for many real-time multimedia and computer vision applications. In previous work, the Harris and KLT algorithms were redesigned to increase the performance by reducing the algorithmic complexity, resulting in the Low Complexity Corner Detector algorithm. To attain further speedup, this paper proposes the implementation of this low complexity corner detector algorithm on a parallel computing architecture, namely a GPU using Compute Unified Device Architecture (CUDA). We show that the low complexity corner detector is 2-3 times faster than the Harris corner detector algorithm on the same GPU platform.**

Keywords-LoCoCo; Harris feature detector; GPU; CUDA

I. INTRODUCTION

High speed detection of feature points is a fundamental requirement of many real-time computer vision and multimedia applications such as image matching, immersive communications, augmented reality, object recognition, mosaicing etc. Among robust and real-time feature point detectors (a good survey can be found in [16]), corner detection algorithms such as Harris [3] and KLT [2] are widely used for their lower complexity compared to SIFT [14] and SURF [15]. Several implementations of corner feature detection algorithms exist on GPUs. Sinha et. al. [4] proposed an implementation of the KLT corner detector for GPUs. However, the last step of non-maximum suppression of the cornerness response was performed on the CPU; this limits the potential speedup that can be obtained by a corner detector algorithm. Teixeira et. al [5] proposed their own implementation of non-maximum suppression for GPUs. They attained a significant speedup but introduced an imprecision of one pixel in the localization of corner points. Moreover, their method used a 3x3 Prewitt filter instead of a 9x9 Gaussian filter to compute image gradients. Both the algorithms [4][5] were implemented using the traditional OpenGL GPGPU API.

For modern GPU architectures, the CUDA [7] framework is supported by Nvidia GPUs for general purpose parallel computing. Compared to traditional GPUs and APIs such as Direct 3D or OpenGL, CUDA provides much more flexibility to manage and utilize GPU resources in order to fully exploit data parallelism in an application. Moreover, CUDA provides a high level programming model and a straightforward method of writing scalable parallel programs to be executed on the GPU. To our knowledge, none of the corner detector algorithm has fully exploited the computational power of CUDA.

Pradip et al. [1] proposed a Low Complexity Corner detector algorithm, which reduces the complexity of the Harris and KLT corner detectors by using a box kernel, integral image, and efficient non-maximum suppression. It achieves a complexity reduction by a factor of 8 on a CPU platform. By exploiting the computation power of CUDA, this paper proposes an efficient mapping of this low complexity corner detector on GPU. The implementation outperforms the execution time of existing state-of-the-art corner detector algorithms on GPUs [4][5].

The remaining of the paper is organized as follows: the low complexity corner detector algorithm is described in Section 2 in order to make the work self-contained. The mapping of the low complexity corner detector on GPU is described in Section 3. Section 4 shows the experimental results and Section 5 concludes the paper.

II. LOCOCO : LOW-COMPLEXITY CORNER DETECTOR

Harris feature detector is based on the local autocorrelation function within a small window of each pixel as shown in (1) and (2), which measures the local change of intensities due to the shifts in a local window:

$$C(\mathbf{p}) = \sum_{\mathbf{x}\in W} \left\{ \begin{bmatrix} g_x^2(\mathbf{x}) & g_x(\mathbf{x})g_y(\mathbf{x}) \\ g_x(\mathbf{x})g_y(\mathbf{x}) & g_y^2(\mathbf{x}) \end{bmatrix} \times v(\mathbf{x}) \right\} = \begin{bmatrix} G_{xx} & G_{xy} \\ G_{xy} & G_{yy} \end{bmatrix} \quad (1)$$
$$\mathbf{x} = (x, y)$$

$$g_i = \partial_i(\mathbf{g} \otimes I) = (\partial_i \mathbf{g}) \otimes I, i \in (x, y) \quad (2)$$

where v(**x**) is a weighting function, which is usually Gaussian or uniform, **p** is a center pixel, W is a window centered at **p**, I is the original image, **g** is Gaussian, and g_x and g_y are the convolution of the Gaussian first order partial derivative with I in x and y directions at point (x, y), respectively.

As shown in (1), the Harris corner detector algorithm computes image derivatives using the Gaussian derivative kernel, computes cornerness response and suppresses non-maximum points to obtain the corner points. LoCoCo reduces the computational complexity of the Harris algorithm in each step. First, by using the integral image and box kernel, the computational cost of gradients is reduced. The box kernel is obtained by approximating the first order Gaussian derivative kernel. Second, many repeated calculations for computation of cornerness according to (1) are reduced by the use of the integral image. Finally, the combination of sorting (to rank cornerness responses) and non-maximum suppression is replaced by the efficient non-maximum suppression [6]. The LoCoCo algorithm is summarized as follows:

1. Calculate the integral image for the original image I.
2. Compute gradients gx and gy by using the integral image and the box kernel approximation.
3. Create the integral images for g2x, g2y and gxy Then, evaluate (1) and (2) and compute the cornerness response. With the use of the integral image, each element of (2) can be evaluated in 4 memory accesses and 3 operations.
4. Efficient non-maximum suppression is performed to suppress the non-maximum point instead of sorting and performing non-maximum suppression.

By following the above mentioned steps, LoCoCo achieves comparable feature detection results and a speedup factor of 8 with respect to Harris on the CPU platform. More details and experimental results are presented in [1].

III. MAPPING LOCOCO USING CUDA

This section explains the mapping of each step of LoCoCo on the GPU using CUDA. In CUDA, the kernel is visualized as a grid, which consists of multiple parallel thread blocks; each thread block can contain up to 512 parallel threads. It is the responsibility of the programmer to choose the number of blocks per grid and the number of threads per block. Once the kernel is launched, the grid blocks are distributed on the parallel multiprocessors as described in [7]. The global memory exists off-chip and is accessible by all threads. The shared memory is on-chip and the threads within a block can communicate and cooperate using the shared memory as well as the thread synchronization mechanism. As described in [12], high performance on CUDA can be achieved by allowing a massive number of active threads to exploit the large number of cores, hence hiding memory latency by computations.

A. Integral Image

LoCoCo makes an extensive use of the integral image. We propose an efficient method to map the computation of the integral image on the GPU. The computation of the integral image can be separated in two stages. As shown in Figure 1(a), the prefix sum is calculated for each row. After completing the processing on rows, as shown in Fig. 1(b), the prefix sum is applied to each column, thus resulting into an integral image.

The prefix sum is computed for all rows in parallel by using the efficient parallel scan algorithm designed for GPUs [8]. The key idea of this algorithm is to divide the block of data into warp-sized chunks and all scan primitives are built upon the set of primitive intra-warp scan routines. The warps execute instructions in SIMD fashion and synchronization is not needed in order to share data within a warp. Thus, the intra-warp scan routine performs scan operations over a warp of 32 threads and computes the prefix sum for 32 elements without requiring any synchronization operation.

The computation of the prefix sum on a row is performed by allocating a thread block to that row and dividing it into warp-sized chunks. All the warps are scanned in parallel using an intra-warp scan routine. Next, the partial results of each scan are accumulated and adjusted to get the scan for the complete row. The reduced number of synchronization steps and various optimizations, such as efficiently exploiting shared memory and performing an initial serial scan of multiple input elements when read from global memory, makes it one of the fastest scans yet designed for the GPU [8].

In order to evaluate the subsequent prefix sum on the columns, the prefix sum result of the rows is transposed and a new row-based scan is launched. Transpose between the two steps help to maintain coalesced access to the global memory [7]. The resultant integral image is not transposed back again to correct the orientation, since the computation of integral image in the subsequent step of computing the cornerness response leads to another transpose, yielding the restored image.

B. Gradients

The strategy used to parallelize this step is based upon creating many threads to exploit the large number of cores. The image is partitioned into a regular grid of blocks. The width and height of these blocks are equal to the 16th of the width and height of the image, respectively. Each thread in that block can be mapped to a pixel location and computes the gradient value corresponding to that pixel of the image. Therefore, for the box kernel, the computations performed by the threads in a block are independent of each other. The CUDA kernel is launched wherein each thread performs eight memory accesses and seven operations in parallel to calculate g_x or g_y, corresponding to each pixel. The step is complete when the gradients are computed for all the pixels in the image.

C. Cornerness Response

The strategy used to compute the integral image for g^2_x, g^2_y and g_{xy} is the same as described in Section 3.1. In order to obtain g^2_x, g^2_y and g_{xy}, the scan algorithm is modified such that each element of g_x and g_y is squared or multiplied with each other when fetched from global memory to shared memory.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-129-8

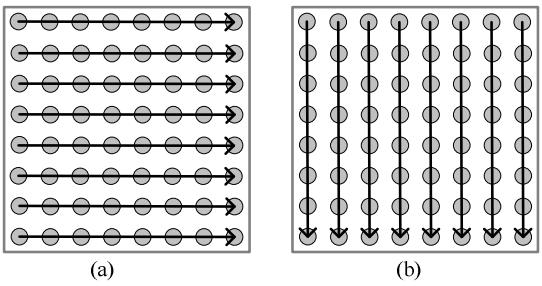

Figure 1. (a) Prefix sum on rows (b) Prefix sum on columns

The computation of the summation within a window and cornerness response corresponding to each pixel, is based on a similar strategy as described in previous section is adopted so that each thread in every block can be mapped to a pixel location in the image. The CUDA kernel is launched wherein each thread performs four memory accesses and three operations in parallel to calculate the window sum corresponding to each pixel followed by the computation of cornerness response. The step is complete when the cornerness response is calculated for all the pixels in the image.

D. Efficient Non-Maximum Suppression

Access to off-chip global memory is slow and requires 200 to 300 cycles per access. This latency can be hidden by launching a massive number of active threads [12]. But this technique does not give enough speedup for algorithms that have repeated calculations or are bounded by memory accesses. Another technique to attain speedup is to use low latency on-chip shared memory and reuse data among all the threads in a thread block to reduce the number of accesses to the global memory [12]. As non-maximum suppression incorporates repeated calculations on a small region of pixels therefore shared memory is exploited to reduce the number of accesses to the global memory.

The suppression algorithm is implemented for a d by d (d=9) neighborhood. The kernel is launched wherein threads in each thread block fetches ($2d$ x $2d$) pixels from global memory to the shared memory. At this point the contents of shared memory can be visualized as four sub-blocks as shown in Figure 2(a). The kernel is implemented in such a way that the threads in a thread block compute the maximum value of the cornerness response in each of the sub-blocks in parallel. These maximum values are termed as candidate local-maximas (max1, max2, max3 and max4). The maximum value in each of the sub-blocks is calculated using parallel reduction [10], where the add operation is replaced with a comparison operator. For each candidate maxima, the threads in a thread block fetch the local neighborhood pixels to the shared memory if the value is greater than the predefined threshold, as shown in Figure 2(b). The maximum value is computed in each of the blocks using [10]. If the candidate maxima remains maximum in the local neighborhood then it is marked as corner point else the point is suppressed.

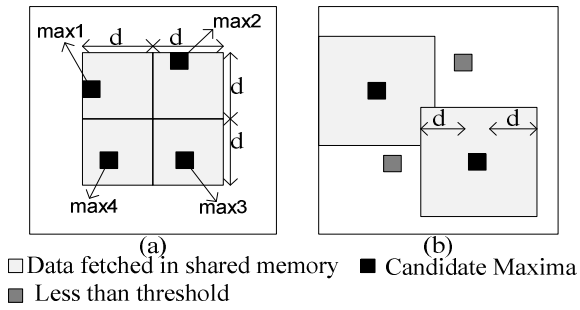

Figure 2. Efficient Non-Maximum Suppression

IV. RESULTS AND DISCUSSION

The execution time of LoCoCo is evaluated on a CPU and on a GPU. To measure the effectiveness of LoCoCo on GPUs, the execution times are compared with our CUDA based implementation of the Harris corner detector on the same GPU.

For Harris, the Gaussian derivative is implemented by using the separable Gaussian convolution kernel [11], which requires less computations compared to the 2D convolution. In order to measure the cornerness response, the gradients are squared and multiplied with each other. Furthermore, the summation within the window is implemented by utilizing the separable convolution. The separable filter can be used to sum the pixels within the window by setting the coefficients of separable filters to 1. This method of implementation runs much faster than computing the naive sum of all pixels within a window. As described in [4], the sorting for cornerness response is performed on the CPU and this involves transferring the complete cornerness image back to the CPU. Instead of adopting this approach, the sorting is performed on the GPU by using an efficient sorting algorithm as presented in [9]. After this step, non-maximum suppression is performed on the GPU. Thus, the Harris algorithm completely runs on the GPU and this implementation is utilized to have a fair comparison with LoCoCo implementation on the GPU.

As shown in Figure 3, the GPU implementation of LoCoCo is around 14 times faster than the corresponding CPU implementation. The speedup is mainly due to the fact that computation of the integral image and efficient non-maximum suppression is efficiently parallelized using CUDA.

The comparison of execution time of both LoCoCo and Harris on GPU is shown in Figure 4 for different image and kernel sizes. As shown in Figure 4(a), for various image sizes and a fixed kernel size of 9x9, the LoCoCo implementation on GPU is around 2 times faster than the Harris corner detector on GPU. The original Harris algorithm uses Gaussian convolution instead of Integral image computation and box kernel approximations; contrary to CPU programming the execution time of the GPU implementation of convolution for small kernel size (9x9) is comparable to the time taken by the computation of the integral image and the box kernel approximation.

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-129-8

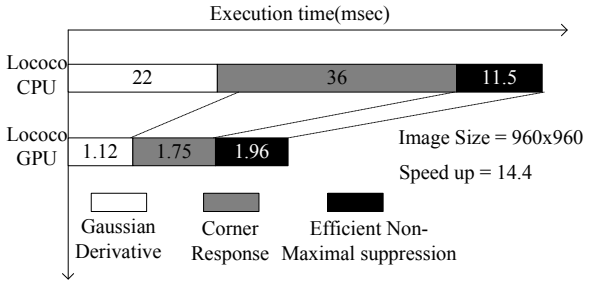

Figure 3. Execution time of LoCoCo on CPU and GPU (CPU: Intel Core 2 Duo E6750, 2.66 GHz and 2 GB RAM, GPU: Nvidia GeForce GTX 280, 1.296 GHz, 1 GB Global memory, 16 KB shared memory per core)

With the box kernel approximation, the speedup for a kernel size of 9x9 is mainly due to the fact that LoCoCo replaces the combination of feature sorting and non-maximum suppression in Harris by efficient non-maximum suppression. As shown in Figure 4(b), for a kernel size of 31x31, LoCoCo is 3 times faster than the Harris. As the kernel size increases, the computation of the integral image and box kernel approximation remains unaffected but the execution time for the convolution increases significantly. For applications that require multi-scale estimation, the convolution must be computed for each scale, while only one execution of the integral image allows for the computation of all the scales. In that case, the LoCoCo algorithm turns out to be much more efficient than the Harris algorithm.

Table 1 presents the comparison of LoCoCo using CUDA with other state-of-the-art implementations of corner detectors. Accelerated corner detector [5] provides a full implementation of the Harris corner detector on GPU by proposing its own version of non-maximum suppression. The proposed non-maximum suppression has two different variants, one in which the cornerness response image is compressed and the other in which the cornerness response image is not compressed. The lossy compression of the cornerness response image introduces a precision error of one pixel in localization of the corner points whereas our method does not introduce any precision error or compression of the cornerness response image. A comparison is made with the version of the accelerated corner detector in which the cornerness response image is not compressed.

The time taken by this implementation is reported in [13]. The timings presented in Table 1 include the time to transfer data between the GPU and the CPU. Notice that execution times reported in [4][5] are related to a GeForce 8800 GTX while the execution times of our contribution have been measured on a GeForce GTX 280. Even though reference [17] indicates that GTX280 delivers twice the performance of GeForce 8800, it can still be inferred that our method is the fastest compared to state-of-the-art implementations of corner detectors reported till now.

TABLE I. COMPARISON WITH OTHER METHODS

Algorithm	Time (ms)	Image Size	Platform
L.Teixeira [5]	7.3	640x480	GeForce 8800 GTX
Sinha [4]	61.7	720x576	GeForce 8800 GTX + AMD Athlon 64 X2 Dual Core 4400 (one core used)
Our Method	2.4	640x480	GeForce 280 GTX

V. CONCLUSION AND FUTURE WORK

In this paper, we proposed an efficient implementation of low complexity corner detector using CUDA. Corner detection using CUDA has not been reported so far. Our method greatly exploits the data parallelism and achieves a speedup factor of 14 with respect to CPU. Experimental result shows that low complexity corner detector is around 2-3 times faster than Harris on a GPU. With the increase of kernel size, the execution time of our method remains close to constant while the execution time of the Harris increases, thus achieving further speedups.

REFERENCES

[1] P. Mainali, Q. Yang, G. Lafruit, R. Lauwereins and L. Van Gool, “LoCoCo: Low Complexity Corner Detector”, ICASSP 2010, pp. 810-813.

[2] C. Tomasi and T. Kanade, “Detection and tracking of point features”, Technical Report CMU, April 1991

[3] C. Harris and M. Stephen, “A Combined corner and edge detector” In Proce. of Alvey Vision Conf., pp. 147-151, 1988

[4] S. Sinha, J. Frahm and M. Pollefeys, “GPU-based video feature tracking and Matching”, in EDGE 2006, workshop on Edge Computing Using New Commodity Architectures, 2006

[5] L. Teixeira, W. Celes and M. Gattass, “Accelerated Corner Detector Algorithms”, in BMVC, 2008

[6] A. Neubeck and L. V. Gool, “Efficient non-maximum suppression”, in ICPR 2006, Vol. 3, pp. 850-855.

[7] http://developer.nvidia.com/object/cuda.html.

[8] S. Sengupta, M. Harris, and M. Garland. “Efficient parallel scan algorithms for GPUs”. NVIDIA Technical Report NVR-2008-003, December 2008

[9] N. Satish, M. Harris, and M. Garland. “Designing efficient sorting algorithms for manycore GPUs”, Proc. 23rd IEEE IPDPS2009, May 2009

[10] M. Harris, “Optimizing Parallel Reduction in CUDA”, NVIDIA Developer Technology

[11] V. Podlozhnyuk, “Image Convolution with CUDA”, Nvidia CUDA 2.0 SDK convolution separable document

[12] S. Ryoo, C. Rodrigues, S. Baghsorkhi, S. Stone, D. Kirk and W. Hwu. “Optimization principles and application performance evaluation of a multithreaded GPU using CUDA”. In Proc. 13th ACM SIGPLAN, 2008.

[13] J. F. Ohmer and N. J. Redding, “GPU-Accelerated KLT Tracking with Monte-Carlo-Based Feature Reselection”, DICTA 2008.

[14] D. Lowe “Distinctive image features from scale-invariant keypoints” International Journal of Computer Vision, 60, 2 (2004), pp. 91-110

[15] H. Bay, A. Ess, T. Tuytelaars, L. Van Gool, "SURF: Speeded Up Robust Features", Computer Vision and Image Understanding (CVIU), Vol. 110, No. 3, pp. 346--359, 2008

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-129-8

[16] T. Tuytelaars, K. Mikolajczyk "Local Invariant Feature Detectors: A Survey", Foundations and Trends in Computer Graphics and Vision, Vol. 3, nb 3, pp 177-280, 2008.

[17] http://www.nvidia.com/content/PDF/fermi_white_papers/N.Brookwood_NVIDIA_Solves_the_GPU_Computing_Puzzle1.pdf

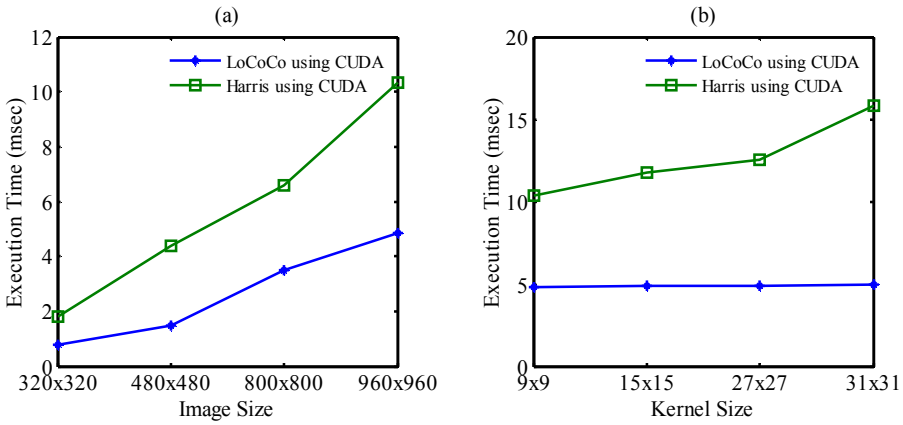

Figure 4. Execution time for LoCoCo and Harris on Nvidia GeForce GTX 280 GPU (a) Comparison for different image sizes (b) Comparison for different kernel sizes

Copyright (c) IARIA, 2011. ISBN: 978-1-61208-129-8