Highly Parallel Implementation of Harris Corner Detector on CSX SIMD Architecture

Fouzhan Hosseini, Amir Fijany, and Jean-Guy Fontaine

Italian Institute of Technology, Genova, Italy
{fouzhan.hosseini,amir.fijany,jean-guy.fontaine}@iit.it

Abstract. We present a much faster than real-time implementation of Harris Corner Detector (HCD) on a low-power, highly parallel, SIMD architecture, the ClearSpeed CSX700, with application for mobile robots and humanoids. HCD is a popular feature detector due to its invariance to rotation, scale, illumination variation and image noises. We have developed strategies for efficient parallel implementation of HCD on CSX700, and achieved a performance of 465 frames per second (fps) for images of 640x480 resolution and 142 fps for 1280x720 resolution. For a typical real-time application with 30 fps, our fast implementation represents a very small fraction (less than %10) of available time for each frame and thus allowing enough time for performing other computations. Our results indicate that the CSX architecture is indeed a good candidate for achieving low-power supercomputing capability, as well as flexibility.

1 Introduction

Mobile robots and humanoids represent an interesting and challenging example of embedded computing applications. On one hand, in order to achieve a large degree of autonomy and intelligent behavior, these systems require a very significant computational capability to perform various tasks. On the other hand, they are severely limited in terms of size, weight, and particularly power consumption of their embedded computing system since they should carry their own power supply. The limitation of conventional computing architectures for these types of applications is twofold: first, their low computing power, second, their high power consumption. Emerging highly parallel and low-power SIMD and MIMD architectures provide a unique opportunity to overcome these limitations of conventional computing architectures. Exploiting these novel parallel architectures, our current objective is to develop a flexible, low-power, lightweight supercomputing architecture for mobile robots and humanoid systems for performing various tasks and, indeed, for enabling new capabilities.

Computer vision and image processing techniques are very common in robotic, e.g. tracking, 3D reconstruction and object recognition. Feature detection is a low-level image processing task which is usually performed as the first step in many computer vision applications such as object tracking [1] and object recognition [2]. Harris Corner Detector (HCD) [3] is a popular feature detector due to its invariance to rotation, scale, illumination variation and image noises.

M.R. Guarracino et al. (Eds.): Euro-Par 2010 Workshops, LNCS 6586, pp. 137–144, 2011.
© Springer-Verlag Berlin Heidelberg 2011

Fast implementations of HCD on various architectures have been considered in the literature including Application Specific Integrated Circuit (ASIC) [4], Field Programmable Gate Array (FPGA) [5], Graphics Processing Units (GPU) [6], and Cell processor [7]. A performance comparison of these implementations is given in Section 5. ASICs and FPGAs could be used to design custom hardware for low-power high performance applications. GPU and Cell processor are more flexible, but the main limitation is the rather prohibitive power consumption. None of the above mentioned solutions satisfies our requirements for mobile system vision processing including low power consumption, flexibility, and real time processing capability simultaneously.

In this paper, we present a fast implementation of HCD on a highly parallel SIMD architecture, the ClearSpeed CSX700. The CSX700 has a peak computing power of 96 GFLOPS, while consuming less than 9 Watts. In fact, it seems that CSX provides one of the best (if not the best) performance in terms of GFLOPS/Watt among available computing architectures. Considering the CSX architecture, we have developed strategies for efficient parallel implementation of HCD. We have achieved a performance of 465 fps for images of 640x480 resolution and 142 fps for 1280x720 resolution. These results indeed represent a much faster than real-time implementation and better than those previously reported in the literature. Our experimental results, presented in this paper, clearly indicate that the SIMD architectures such as CSX can indeed be a good candidate for achieving low-power supercomputing capability, as well as flexibility, for embedded applications.

This paper is organized as follows. In Section 2, we briefly discuss the HCD algorithm. In Section 3, we briefly review the CSX architecture. In Section 4, our approach for parallel implementation of HCD on CSX architecture is described and experimental results are discussed in Section 5. Finally, some concluding remarks are presented in Section 6.

2 The Harris Corner Detector Algorithm

To detect corners in a given image, the HCD algorithm [3] proceeds as follows. Let $I(x, y)$ denotes the intensity of a pixel at row x and column y of the image.

1. For each pixel (x, y) in the input image compute the elements of the Harris matrix $G = \begin{bmatrix} g_{xx} & g_{xy} \\ g_{xy} & g_{yy} \end{bmatrix}$ as follows:

$$g_{xx} = \left(\frac{\partial I}{\partial x}\right)^2 \otimes w \quad g_{xy} = \left(\frac{\partial I}{\partial x}\frac{\partial I}{\partial y}\right) \otimes w \quad g_{yy} = \left(\frac{\partial I}{\partial y}\right)^2 \otimes w, \qquad (1)$$

   where $\otimes$ denotes convolution operator and w is the Gaussian filter.
2. For all pixel (x, y), compute Harris' criterion:

$$c(x, y) = \det(G) - k(trace(G))^2 \qquad (2)$$

   where $\det(G) = g_{xx} \cdot g_{yy} - g_{xy}^2$, k is a constant which should be determined empirically, and $trace(G) = g_{xx} + g_{yy}$.

3. Choose a threshold τ empirically, and set all $c(x, y)$ which are below τ to 0.
4. Non-maximum suppression, i.e. extract points (x, y), which have the maximum $c(x, y)$ in a window neighborhood. These points represents the corners.

3 The CSX700 Architecture

In this section, we briefly review the ClearSpeed CSX700 architecture with emphasis on some of its salient features that have been exploited in our implementation (see, for example, [8] for more detailed discussion). As illustrated in Fig. 1(a), CSX700 has two similar cores, each core has a DDR2 memory interface and a 128KB SRAM, called external memory. Each core also has a standard, RISC-like, control unit, also called *mono execution unit*, which is coupled to a highly parallel SIMD architecture called *poly execution unit*.

Poly execution unit consists of 96 processing elements (PEs) and performs parallel computation (see Fig. 1(b)). Each PE has a 128 bytes register file, 6KB of SRAM, an ALU, an integer multiply-accumulate (MAC) unit, and an IEEE 754 compliant floating point unit (FPU) with dual issue pipelined add and multiply. The CSX700 has clock frequency of 250 MHz [9]. Considering one add and one multiply floating point units working in parallel and generating one result per clock cycle, the peak performance of each PE is then 500 MFLOPS, leading to a peak performance of 96 GFLOPS for two cores (one chip). However, sequential (i.e., scalar) operations, wherein single add or multiply is performed, take 4 clock cycles to be performed [9]. This results to a sequential peak performance of 12 GFLOPS for two cores. This indeed represents a drastic reduction in the peak, and hence, achievable performance. However, vector instructions which operate on sets of 4 data are executed much faster, e.g, vector add or multiply instructions take 4 cycles to be completed [9]. Therefore, vector instructions allow greater throughput for operations. However, the code generated by compiler may not be optimized. Therefore, in order to achieve the best performance, we have also written part of our codes in assembly language of the CSX.

Poly execution unit includes a Programmable I/O (PIO) unit (Fig. 1(b)) which is responsible for data transfer between external memory and PEs' memories, called poly memory. The architecture of poly execution unit enables the computational units and the PIO unit to work in parallel, i.e. it allows overlapping of communication with computation. This feature is fully exploited in our implementation to reduce I/O overhead

Moreover, as shown in Fig. 1(b), a dedicated bus called *swazzle path* connects the register files of neighboring PEs. Consequently, on each cycle, PEs are able to perform a register-to-register data transfer to either their left or right neighbor, while simultaneously receiving data from the other neighbor.

4 Proposed Parallel Implementation

Considering the SIMD architecture of CSX, we have employed data parallel model of computation. Here, we first discuss our data decomposition strategy. Then, we discuss more details of our parallel implementation.

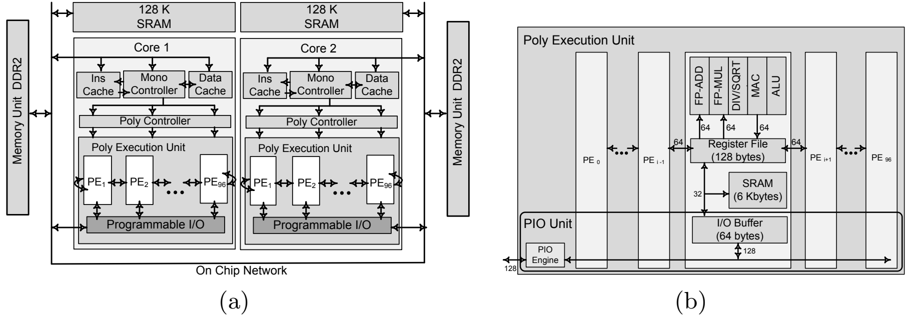

Fig. 1. (a)Simplified CSX Chip Architecture (b) Poly Execution Unit Architecture [8]

4.1 Data Decomposition

Having an image and an array of PEs, various data distributions schemes could be considered. The most obvious schemes are row (column)-stripe distribution, block distribution, and row (column)-cyclic distribution. Here, we discuss the effectiveness of each of these data distribution schemes for parallel implementation of HCD algorithms on the CSX architecture. An important consideration for the CSX architecture is the size of PE's memory which is rather limited. For the CSX architecture, various data distributions should be compared in terms of the following parameters: (a) required memory space for each PE; (b) redundant external memory communication; and (c) inter-PE communication time.

In the following, c and r denote the number of columns and rows in image matrix, respectively. According to the algorithm description in Section 2, HCD performs a set of operations in windows around each pixels. In fact, HCD uses windows which may have different sizes in 3 stages: calculating partial derivatives, Gaussian smoothing, and non-maximal suppression. Let ω be the sum of these window sizes. Also, let p indicate the number of PEs. Finally, in each memory communication, each PE reads or writes m bytes of data (pixel) from/into the external memory. Π is the memory space needed to calculate the elements of Harris matrix for m pixels.

Block distributions. In this scheme, as illustrated in Figure 2(a), the image is divided into $p = d * s$ blocks, with each block having c/d columns and r/s rows. The first block is assigned to the first PE, the second one to the second PE, and so on. Each block can be identified by an ordered pair (i, j) where $1 \leq i \leq s$ and $1 \leq j \leq d$. In the following, $P(i, j)$ denotes the PE which is responsible for processing the block (i, j) and refers to PE $((i - 1)s + j)$.

Figure. 2(b) depicts the boundary data needed for computation by $P(i, j)$ and its four immediate neighbors. To handle boundary data, needed by two neighboring PEs, there are two possible alternatives: transferring boundary data from external memory to both PEs, hence performing redundant data communication, or transferring to one PE and then using swazzling path to transfer it to the other PE. The former takes more time, and the latter requires more PE memory

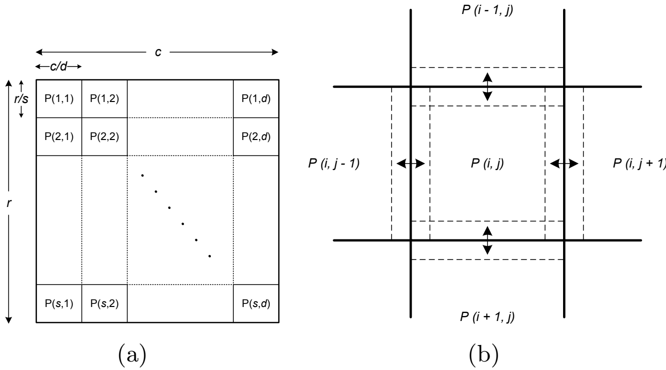

Fig. 2. (a)Block distribution. P(i, j) refers to $PE(i-1)s+j$ (b) Boundary data for each PE in block distribution.

space to store boundary data as discussed in the following. To process first rows (columns), $P(i,j)$ requires the last rows (columns) of $P(i-1,j)$ ($P(i,j-1)$), but these PEs have not yet received the data which $P(i,j)$ requires. Therefore, if the swazzling path is used then $P(i,j)$ should skip processing these boundary data, until $P(i-1,j)$ ($P(i,j-1)$) provides the required data. Also, for processing the last rows (columns) of data, $P(i,j)$ requires data which has already been sent to $P(i+1,j)$ ($P(i,j+1)$). For these PEs to provide the boundary data to $P(i,j)$, they need to store this part of data in their memory which is a limited resource. It should be noted that on the CSX architecture, the distance between $P(i,j)$ and $P(i+1,j)$ which process two neighboring blocks is d.

Row-strip distribution. The first r/p rows are assigned to the first PE, the second r/p rows are assigned to the second PE, and so on. To handle boundary data, $PE(i)$ requires last rows of $PE(i-1)$ and first rows of $PE(i+1)$. In fact, like block distribution, boundary data could be transfered from external memory to both PEs or from one PE to another via swazzling path. As discussed above, the former takes more time, and the latter requires more PE memory space.

Row-cyclic distribution. In this scheme, the first row is assigned to the first PE, the second row to the second PE, and so on. Since one row is assigned to each PE, each PE needs to communicate with the PEs which are at most at the distance of $(w-1)/2$. Here, each PE needs data just after its neighbor has finished processing that same data. So, swazzle path can be utilized without using extra poly memory space.

Table 1 summarizes the parameters calculated for each data distribution strategy. As can be seen, block and row-strip distribution schemes require either more PE memory space or more redundant external memory communications. In fact, for these schemes, the required poly memory space increases linearly with ω. Note that, the size of windows in HCD are determined empirically for each application. For larger ω, e.g. 7 or 11, using these data distributions, the required PE memory will be larger than poly memory space. Row-cyclic distribution needs less poly

Table 1. Figure of merit for different data distribution schemes. S indicates that boundary data is shared between PEs by using swazzling path. M indicates that boundary data is transferred from external memory.

Data Dist.		Redundant External Memory Comm.	Inter-PE Comm.	PE Memory Space
Block Dist.	M	$cs(\omega-1)$	$r(\omega-1)$	$\omega\Pi+m$
	S	-	$(\omega-1)[cs+r]$	$(\omega+\frac{\omega-1}{2})\Pi+m$
Row-strip Dist.	M	$pc(\omega-1)$	-	$\omega\Pi+m$
	S	-	$c(\omega-1)$	$(\omega+\frac{\omega-1}{2})\Pi+m$
Row-cyclic Dist.		-	$c\frac{\omega(\omega-1)}{2}$	$\Pi+m$

memory space and no redundant external memory communication. Although row-cyclic distribution uses inter-PE communication more than row-strip distribution by a factor of $\omega/2$, this overhead will be negligible since communication via swazzle path is very fast (see Section 3). Therefore, row-cyclic distribution scheme is the most efficient for implementing HCD on the CSX architecture.

4.2 Parallel Implementation of Harris Corner Detector Algorithm

In this section, we discuss parallel implementation of HCD on the CSX architecture, based on row-cyclic distribution scheme. Since, each CSX core includes 96 PEs, the input image is divided into groups of 96 rows. The computation of each group represents a sweep and sweeps are performed iteratively. Also, to utilize both cores of CSX700 processor, the input image is divided into two nearly equal parts. The first $\lceil r/2\rceil+(\omega-1)/2$ rows are assigned to the first core and the last $\lfloor r/2\rfloor+(\omega-1)/2$ rows are assigned to the second core. Sending boundary lines to both cores enables each core to perform all computation locally. In the following, implementation of HCD on one core is explained (for one sweep).

Memory Communication Pattern. In our parallel implementation, communication and computation overlapping is greatly exploited, and PEs are never idle to receive data (except the initial phase) from external memory. In fact, each image row is divided into segments of almost equal size (32 or 64 pixels, depending on the image size). After receiving the first segment of data, while each PE is processing the segment of data which is already in its memory, in the background, PIO transfers new sets of data from external memory to memories of PEs and the last sets of results to external memory.

Computation Steps. In this section, we present the computation of one segment of data which consists of 5 steps: calculating partial derivative of I in x and y directions, Gaussian smoothing, computing Harris criterion, non-maximum suppression, followed by thresholding.

To calculate partial derivative of I, we have used Prewitt operator. Prewitt operator uses two 3x3 kernels which are convolved with the original image. In our implementation, we take advantages of the fact that these convolution kernels are separable, i.e. they can be expressed as the outer product of two vectors.

So, the x and y derivative can be calculated by first convolving in one direction (using local data), then swazzling data and convolving in the other direction.

In the next step, Gaussian smoothing, elements of Harris matrix, g_{xx}, g_{xy}, and g_{yy} are calculated using (1). Since Gaussian kernel is also separable, the 2-D convolution can be performed by first convolving with a 1-D Gaussian in the x direction, and then swazzling the calculated values and convolving with another 1-D Gaussian in the y direction. Then, Harris' criterion is computed using (2).

In the next step, non-maximum suppression, the maximum value of Harris criterion in each 3x3 neighborhood is determined. First, each PE obtains the maximum value in 1x3 neighborhood. Then, each PE swazzles the maximum values to both its neighbors. Receiving the maximal values of two neighboring rows, the maximum value in 3x3 neighborhood can then be obtained.

5 Results and Performance of Parallel Implementation

To evaluate the performance, we have implemented the following HCDs on the CSX700 architecture: $HCD_{3\times3}$ and $HCD_{5\times5}$ which uses a 3×3 and 5×5 Gaussian kernel, respectively. Since our proposed parallel approach provides flexibility, it can be easily applied to images with different sizes, and various sizes of Gaussian filter or non-maximum suppression window. The performance of implemented algorithms in terms of latency, fps, and sustained GFLOPS for different image resolutions are summarized in Table 2. As Table 2 shows, for all tested image resolutions, even for resolution of 1280x720, our implementation is much faster than real-time.

The arithmetic intensity, i.e., number of operation per pixel, of $HCD_{3\times3}$ and $HCD_{5\times5}$ is 40 and 64 respectively. As Table 2 shows, the sustained GFLOPS depends also on the image size. One reason is that in processing the last sweep of data, some PEs may be idle, and the number of idle PEs depends on image size.

Table 2. Performance of HCD on CSX700 architecture using 3×3 and 5×5 Gaussian filter

Image	Latency (ms)		fps		Sustained GFLOPS	
Resolution	$HCD_{3\times3}$	$HCD_{5\times5}$	$HCD_{3\times3}$	$HCD_{5\times5}$	$HCD_{3\times3}$	$HCD_{5\times5}$
128x128	.165	.224	6060	4464	3.97	4.68
352x288	.8	1.22	1250	819	5.06	5.31
512x512	1.74	2.63	574	380	6.02	6.37
640x480	2.15	3.28	465	304	5.71	5.99
1280x720	7.04	10.89	142	91	5.23	5.41

Table 3. Comparison with other implementations in the literature

Image Resolution	fps reported in [ref]	fps achieved by our approach
128x128	1367 [4]	4464
352x288	60 [5]	819
640x480	99 [6]	304

Table 3 compares our implementation results with those reported in the literature. As can be seen, our approach provides much better performance in terms of latency or frame per second while providing a high degree of flexibility in terms of problem size and parameters.

6 Conclusion and Future Work

We presented a much faster than real-time implementation of Harris Corner Detector (HCD) on a low-power, highly parallel, SIMD architecture, the ClearSpeed CSX700. Considering the features of the CSX architecture, we presented strategies for efficient parallel implementation of HCD. We have achieved a performance of 465 fps for images of 640x480 resolution and 142 fps for 1280x720 resolution. These results indeed represent a much faster than real-time implementation. Our experimental results, presented in this paper, and our previous work [10] clearly indicate that the CSX architecture is indeed a good candidate for achieving low-power supercomputing capability, as well as flexibility, for embedded computer vision applications.

References

1. Yilmaz, A., Javed, O., Shah, M.: Object tracking: A survey. ACM Computing Surveys 38(4), 13 (2006)
2. Roth, P.M., Winter, M.: Survey of appearance-based methods for object recognition. Technical Report ICG-TR-01/08, Inst. for Computer Graphics and Vision, Graz University of Technology (2008)
3. Harris, C., Stephens, M.: A combined corner and edge detector. In: 4th Alvey Vision Conference, pp. 147–151 (1988)
4. Cheng, C.C., Lin, C.H., Li, C.T., Chang, S.C., Chen, L.G.: iVisual: an intelligent visual sensor SoC with 2790fps CMOS image sensor and 205GOPS/W vision processor. In: 45th Annual Design Automation Conference (DAC 2008), pp. 90–95 (2008)
5. Dietrich, B.: Design and implementation of an FPGA-based stereo vision system for the EyeBot M6. University of Western Australia (2009)
6. Teixeira, L., Celes, W., Gattass, M.: Accelerated corner-detector algorithms. In: 19th British Machine Vision Conference(BMVC 2008), pp. 625–634 (2008)
7. Saidani, T., Lacassagne, L., Bouaziz, S., Khan, T.M.: Parallelization strategies for the points of interests algorithm on the cell processor. In: Stojmenovic, I., Thulasiram, R.K., Yang, L.T., Jia, W., Guo, M., de Mello, R.F. (eds.) ISPA 2007. LNCS, vol. 4742, pp. 104–112. Springer, Heidelberg (2007)
8. ClearSpeed. Clearspeed Whitepaper: CSX Processor Architecture (2007), `http://www.clearspeed.com`
9. ClearSpeed: CSX600/CSX700 Instruction Set Reference Manual, 06-RM-1137 Revision: 4.A (August 2008), `http://www.clearspeed.com`
10. Hosseini, F., Fijany, A., Safari, S., Chellali, R., Fontaine, J.G.: Real-time parallel implementation of SSD stereo vision algorithm on CSX SIMD architecture. In: Bebis, G., Boyle, R., Parvin, B., Koracin, D., Kuno, Y., Wang, J., Wang, J.-X., Wang, J., Pajarola, R., Lindstrom, P., Hinkenjann, A., Encarnação, M.L., Silva, C.T., Coming, D. (eds.) ISVC 2009. LNCS, vol. 5875, pp. 808–818. Springer, Heidelberg (2009)