A novel embedded system for vision tracking

Antonis Nikitakis, Theofilos Paganos
Technical University of Crete,
Department of Electronic and Computer Engineering
Kounoupidiana, Chania, Crete, GR73100, Greece
anikita@mhl.tuc.gr; pagantheo@gmail.com

Ioannis Papaefstathiou
Synelixis Solutions Ltd,
Farmakidou 10,Chalkida, GR34100, Greece
ygp@synelixis.com

Abstract— One of the most important challenges in the field of Computer Vision is the implementation of low-power embedded systems that will execute very accurate, yet real-time, algorithms. In the visual tracking sector one of the most promising approaches is the recently introduced OpenTLD algorithm which uses a random forest classification method. While it is very robust, it cannot be efficiently parallelized in its native form as its memory access pattern has certain characteristics that make it hard to take advantage of the conventional memory hierarchies. In this paper, we present a novel embedded system implementing this algorithm. We accelerate the bottleneck of the algorithm by designing and implementing a high bandwidth distributed memory sub-system which is independent of the various software parameters. We demonstrate the applicability and efficiency of this novel approach by implementing our scheme in a modern FPGA.

Keywords— Embedded System, Object Tracking, Distributed Memory, Random Forests, Classification, FPGA.

I. INTRODUCTION

The problem of long term visual tracking is very important in numerous application domains including surveillance, security, augmented reality and multimedia."Long-term" object tracking refers to circumstances where there are large video sequences that contain frame-cuts and fast camera movements and thus the object may temporarily disappear from the scene. The OpenTLD [5] algorithm tries to address the two most critical points in all those applications which are the accuracy of the system as well as its real time performance at high resolutions.

This paper presents a novel embedded memory subsystem allowing for the efficient implementation of the OpenTLD algorithm. Our approach is tailored to reconfigurable devices, as they provide an excellent way to customize the scale and the parameters of the scheme in order to adapt to different applications and cost demands. To support this case we show that our scheme significantly accelerates the underlying algorithm in a "transparent way" allowing the user to change any of the classification parameters seamlessly. The proposed architecture, when prototyped on a low-cost FPGA which incorporates a dual core ARM CPU, can execute this high-end tracking algorithm at a rate of more than 20 times higher than that triggered when a modern dual-core CPU executes the exact same detection scheme.

II. RELATED WORK

[8] presents the implementation of an object recognition system based on the Random Forest classifier and targeted to an FPGA platform. The specified approach utilizes a Logarithmic Number System while their architecture executes the algorithm in an optimized, yet sequential manner. The author assumes that everything is on on-chip memories and can be accessed in a single cycle whereas he does not present any performance (i.e. latency or bandwidth) results nor he tries to parallelize the classifier in any manner. In [2] the authors present a hardware architecture that implements the most compute-intensive part of the OpenTLD object tracking algorithm. They exploit another approach for parallelization which is inherent in the random forest classifiers: they access the forest trees simultaneously. As all decision trees are processing the same input they use multiple copies of the same image data (within a sub-window) for each tree. In comparison with the systems presented above, our approach is the only one which is application-independent whereas, to the best of our knowledge, this is the first embedded scheme that allows for the efficient parallelization of the Random Forest classification problem for computer vision applications, while being significantly faster and more energy efficient than the existing hardware approaches.

III. THE OPENTLD ALGORITHM

The authors of [5] investigate the problem of robust, long-term visual tracking of unknown objects in unconstrained environments. In the object detection part of their tracker implementation, they use a sequential randomized forest classifier [1]. The forest consists of several trees, where each of them is built from a certain group of randomly sampled features. When trying to efficiently implement the algorithm in hardware, although there is plenty of inherent parallelism, it is very hard to utilize a memory interleaving scheme without duplicating data, as they do in [2], due to the randomized access pattern in the Random Forest classifier. As proven by various profiling tests we have performed in an Intel state-of-the-art CPU, the main task of the algorithm, which is the one performing the actual feature detection and involves the Random Forest classifier, spends about 90% of its execution time in memory related sub-tasks (i.e. memory accesses and addressing); this proves that the OpenTLD scheme is very memory intensive. As a reference software implementation we used the one in [3] which has been the first published implementation of this scheme in C++.

IV. HARDWARE ARCHITECTURE

Most of the hardware accelerators, for computer vision tasks, try to efficiently un-roll a big loop and then, by using a memory interleaving scheme, try to exploit the possible

978-3-9815370-2-4/DATE14/©2014 EDAA

inherent parallelism. Our approach is to re-distribute the memory items so as to allow for numerous parallel memory accesses. Thus we applied a 1-1 scrambling process in the memory addressing that enabled us to reduce this collision rate drastically. Adaptive interleaving memory techniques have been already proposed in FPGA-devices such as in [6] and [7] but none of them is targeting the computer vision domain, whereas none of them can be applied to our system.

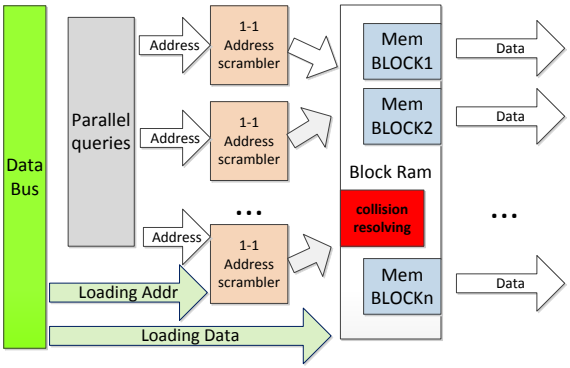

Figure 1. Random parallel queries referring on different block rams

Figure 1 shows our scrambling scheme which is applied during the memory loading process: Our system scrambles the addresses in such a way that the concurrent memory accesses are allocated to different memory sub-blocks. During the query process the address scrambler is applied again in order to re-map the actual addresses into our "scrambled" address space and retrieve the correct data. Furthermore, the scrambling process should be properly selected so as to maximize the parallel accesses.

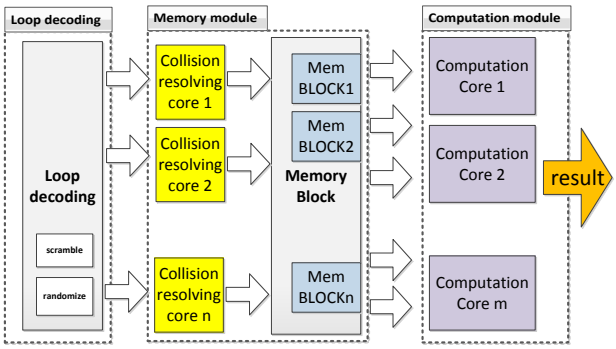

Figure 2. High level architecture

Moving to the high-level architecture of our system (shown in Figure 2), which utilizes our novel memory scheme, it comprises of three basic modules, the **Loop Decoding module**, the **Memory module** and the **Computation module**.

A. Loop decoding module

In this module we actually decode the main loop of the OpenTLD scheme, which at the lowest grain-level implements the Random Forest memory accesses. In particular, it implements the sliding window movement function as well as the randomized sampling method which uses predefined coefficient matrices. The loop decoding module produces 32 memory accesses per cycle after decoding 8 continuous iterations of the inner loop of the original OpenTLD code.

B. The Memory Module

This is the main module of our system and as the memory accesses are random and should be performed in parallel, we have to match the query addresses to the available memory blocks in an optimal manner. In order to achieve that a collision detection module monitors whether there are any colliding memory accesses. The non-colliding accesses are routed to the corresponding memory block while the colliding ones are serialized appropriately. The serialization module comprises of a series of simple finite state machines (in a cascade interconnection) and can serialize a single memory access in every clock cycle. Obviously, the collisions in one block do not affect in any way the accesses in the other blocks; in other words we may end up with an out-of-order execution of the memory accesses but this does not affect, by nature, the correctness of the Random Forest classifier in any way.

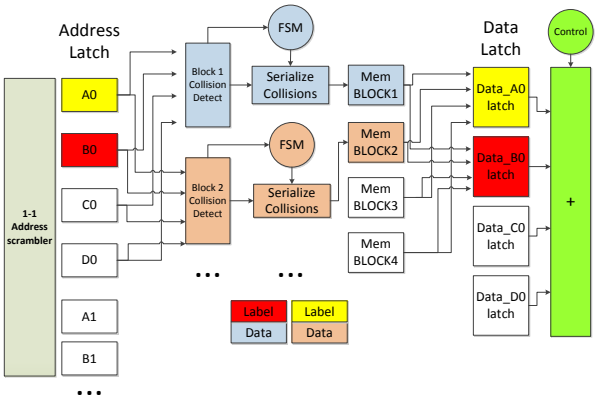

Figure 3. Memory module microarchitecture

Figure 3 demonstrates the dedicated collision resolving module which is attached to each block and which processes all the address queries simultaneously (we draw only 4 queries in this figure for simplicity reasons). The A0,B0,C0,D0 latches are referring to 4 address registers which hold the 4 different memory addresses until the query is completed. As the input addresses, which are kept in the address registers, can refer to any of the available memory blocks we have to label them (shown with different colors in Figure 3) in order to track the corresponding data when those are sent from the Memory module to the Computation module (the one shown in green in Figure 3), since this can be performed out-of-order.

Initially and in order to verify our novel memory approach we have utilized it in a real-world experiment. In particular, we have carefully studied on real-data how the sequential memory accesses are distributed to each memory block with and without our scrambling. We assumed 32 equally distributed memory blocks for a 640x480 frame. Each block contains about 10K addresses while the complete frame requires about 300K. Reordering the address bits in hardware is a simple and zero latency function that redistributed the previously neighboring addresses into different blocks as proved by our experimental results. We also used real-world memory access patterns for measuring the performance of our system. In Table 1 we present a statistical analysis over the total memory accesses ($6.81 * 10^6$) needed for the object detection on a single frame. The average collision metric shows how many

queries per loop execution are referring to this same memory block.

Table 1. Collision rate statistical analysis

Memory Distribution in blocks	Average collisions without scrambling	Average collisions 1-1 scrambling	Memory access per cycle (dual port)
8	6.99	2.65	6.06
16	14.98	3.15	10.16
32	24.5	**3.6**	**17.78**

Those results clearly demonstrate that, if a conventional memory distribution scheme is used, increasing the number of blocks does not result in any significant increase in the performance since many of the memory accesses target the same block. In other words the classification scheme cannot be parallelized efficiently. After applying our scrambling function we reduced the average collision rate to 3.6 collisions per 2x32 memory accesses while the standard deviation is very small (we have up to 4 collisions in the worst case). As a result by using our very simple scrambling scheme and a distributed memory we are able to perform 17.78 memory accesses per cycle (in average) using 32 dual-port on-chip memory blocks.

C. The Computation module

The Computation module is taking the data output of the Memory module and performs the feature computation function; in our case it is a simple sum (i.e. A-B-C+D). It also takes as input the labels corresponding to each data item as the operand's data can be transferred on any of the 32 memory output ports (shown in Figure 3 with green). After the application specific computation is executed the necessary, in every Random Forest classifier, a likelihood table (i.e Leaf Posterior) memory lookup is performed in order to determine if the examined object is part of the trained dataset or not. The update of the Leaf Posterior likelihood table is a relative easy task since only a small number of entries are updated at the frame-processing stage.

V. Evaluation and Performance Results

A. Performance Results

In order to evaluate the performance of the proposed scheme we have executed the same application in a state-of-the-art 2.4GHz dual core CPU and on our targeted FPGA which was clocked at a moderate 200MHz clock. The Intel CPU executed the detection process in 80.4 msec in average (the variation is very small), when only one core has been utilized. Moving to our hardware system and starting from a single memory block and gradually increasing the number of them (i.e. created a distributed memory) while utilizing our novel memory scrambling scheme, we ended up with a considerable speedup over the Intel CPU. Table 3 clearly highlights that our performance grows linearly with the number of blocks and this is due to our very simple, yet very efficient memory scrambling module. Based on our measurements our system performs the detection task in 1.92msec, in average, for each frame while the variation is negligible. This number does not include any I/O overhead between the FPGA and a CPU which will perform the pre-processing tasks as well as the visualization of the results. In the next subsection we describe the complete autonomous embedded system and in this case the I/O is also taken into account.

Table 3. Performance evaluation on a Virtex-6 VLX130T at 200Mhz

Memory partitioning in blocks (dual port)	Average collisions (with scrambling)	Speedup @ 200 Mhz vs Single Core CPU	Effective Memory BW (GB/sec)
1	32	0.96	0.29
8	2.65	14.64	3.54
16	3.15	23.27	5.95
32	**3.6**	**41.98**	**10.42**

The measured speedup, against a dual-core implementation, triggered by our FPGA approach when 32 dual ported block memories were utilized, is 26x; as a result we demonstrate that even the dual channel memory system of a state-of-the-art CPU featuring 3MB of L3 cache cannot outperform our approach which relies on the 10GB/sec average measured bandwidth that our distributed memory system is providing.

B. Embedded design and communication cost

For the implementation of the complete embedded system (the Virtex-6 implementation of the last subsection covers only the actual core of the OpenTLD which is the detection task and not the pre-processing and the visualization of the results) we utilized the very low-cost Avnet's Zedboard [9] which is powered by a Xilinx Zynq-7000 SoC (XC7Z020).

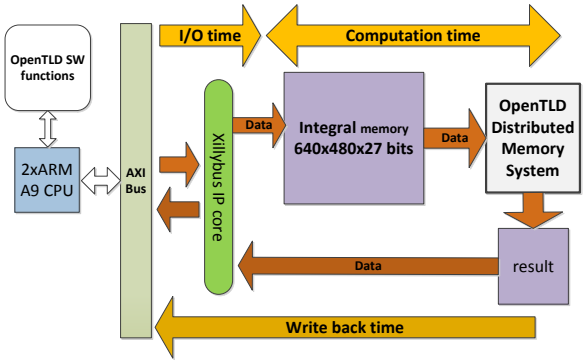

Figure 4. Xilinx Zynq-7000 verification scheme

Since we could not fit a complete 640x480 integral image in the memory blocks of this low-cost device, we have used a smaller image size (480x360) in order to measure the real world performance and then we projected the measurements to the 640x480 frame size initially used. Our test platform was set around a standard Linux distribution for ARM, loaded with the latest OpenCV library. The connection between the dual-core ARM and the reconfigurable resources has been realized through the Xillybus IP core [10]. For our experiments we set the bus clock at 100MHz while our hardware scheme works at 200MHz. In our proposed embedded architecture, demonstrated in Figure 4, and in order to minimize the inter-communication overhead we chose to suppress the amount of data we had to move to the hardware side. This was possible by generating the integral image on-chip during loading and thus instead of moving 640 x 480x27 bits (i.e. size of integral image) from the CPU to the FPGA fabric we had to move only 640x480x8 bits (i.e. size of the original image); this triggers an almost 4 times reduction. When our system handled a 480x360

image we measured the bus bandwidth to be 200Mb/sec. This is much lower than the 370Mb/sec official bandwidth reported by Xillybus, when working at 100MHz, and it was due to a certain limitation of the DMA controller of the specific device used. Based on those measurements we need 1.46msec for the loading of a 640x480 image on the current configuration which will be lowered to 0.79msec when the problem with the DMA controller is addressed. The hardware processing time is 1.92msec in average as mentioned in subsection B, so if we use a double buffering scheme (i.e. using smaller images or utilizing a larger device) we can hide the communication latency completely. Even if no such scheme is utilized and we have to add the intercommunication overhead to the hardware processing overhead our system will still be more than 12x faster than a dual core Intel CPU.

C. Comparison with a GPU implementation

In order to further prove the efficiency of our approach we have implemented the OpenTLD algorithm in a highly parallel GPU platform. In this experiment we also have a Host CPU which executes the complete algorithm except for the detect() function which is executed on the GPU device. The GPU utilized is the NVidia GTX 285 device and it is programmed using the well established CUDA API [11]. In Table 4 we summurize the performance of the reference single core CPU, the GPU and our embedded system.

Table 4: Performance Evaluation in terms of speedup

Accelerated Entity	Software single-core (msec)	CUDA (msec)	CUDA vs single-core	Embedded (msec)	Embedded vs single-core
Detect() with I/O	80.4	7.5	**10.6x**	3.38	**23.78x**
Detect() no I/O	80.4	3.05	**26.36x**	1.92	**41x**

The above results clearly demonstrate that our embedded system outperforms the GPU by at least a 2x factor when the I/O overhead is also taken into account. This speedup will be much higher when the DMA controller problem with the FPGA device used is addressed (3x speedup) or a double buffering scheme is utilized (4x speedup). Even without taking the I/O improvement into account our embedded device can perform the actual processing at a higher rate than a modern GPU. Moving to the energy consumption the Intel CPU has a nominal power consumption of 14W when one core is utilized and the GPU consumes more than 85W, while our system consumes at most 4W. Given the speedup triggered by our approach, our novel embedded device consumes at least 40x less energy than either the CPU or the GPU when executing the openTLD complete applications.

D. Comparison with existing hardware schemes

Moving to the comparisons with the existing hardware approaches, to the best of our knowledge, there is none that has implemented the Random Forest trees structures in hardware in such a generic, not application-specific, way. In particular, the scheme proposed in [2] is likely to induce serious issues as the number of cores scales since: a) each local cache applied to a single core is fed from the same on-chip image memory or the same fully shared local bus, and b) local caches are increasing the on-chip memory usage in a linear way to the number of cores even in the case that the image is stored off-chip. The authors assume that they can easily supply data to the 20 different classifiers that can fit on an FPGA from a single on-chip integral memory (no further details about it are given); however, since the problem is memory bound further studies are needed in order to investigate whether the specified memory bandwidth can indeed be supplied by a single on-chip memory module. More importantly, our approach has certain advantages when compared with the one in [2]: a) our system can support any combination of forest trees and classification features without changing a single wire in our hardware scheme as the statistical characteristics of memory accesses have not been affected, b) our system architecture is not setting any restriction in the sub-window size such as in [2] and c) in our case the number of processing cores can be increased without any need to increase the number of memory blocks. We just split the existing memory in more slices and get a sub-linear bandwidth increase as shown in Table 3.

VI. Conclusions

In this paper we present a simple, yet effective, distributed memory sub-system, upon which we efficiently parallelize and implement, as an autonomous embedded system, the popular OpenTLD tracking scheme. Our real-world measurements demonstrate that the speedup achieved by our embedded system over a modern multi-core CPU is more than 23x while our device is even faster than a highly parallel GPU. Moreover, our system consumes more than 40x less energy than the CPU and the GPU. Since our approach is also very flexible, modular and low-cost, it can be efficiently utilized in numerous multimedia applications which involve the Random Forest approach.

Acknowledgment

This work was supported by the Greek General Secretariat of Research and Technology's grant Aristia-2427 (AFORMI).

References

[1] BREIMAN LEO. 2001. Random Forests. In International Journal of Machine Learning ,Volume 45 Issue 1.

[2] BECKER T., LIU Q., LUK W., NEBEHAY G., AND PFLUGFELDER R.. 2011. Hardware-accelerated object tracking. In Proc. Int. Conf. on Field Programmable Logic and Applications (FPL), Sept. 2011.

[3] BPTLD.2011. https://github.com/Ninjakannon/BPTLD.git

[4] VIOLA PAUL, JONES MICHAEL. 2001. Rapid Object Detection using a Boosted Cascade of Simple Features. Int. Conf. on Computer Vision and Pattern Recognition.

[5] KALAL ZDENEK, MATAS JIRI, MIKOLAJCZYK KRYSTIAN. 2009. Online learning of robust object detectors during unstable tracking. In 3rd On-line Learning for Computer Vision Workshop, Kyoto, Japan, IEEE CS.

[6] TOM VANCOURT AND MARTIN C. HERBORDT.2006. Application-Specific Memory Interleaving Enables High Performance in FPGA-based Grid Computations. In FCCM '06 Proc. of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, Pages 305-306.

[7] M. B. GOKHALE AND J. M. STONE. 1999. “Automatic Allocation of Arrays to Memories in FPGA Processors With Multiple Memory Banks.” Proc. FCCM 1999

[8] OSMAN, H.E. 2009. Random forest-LNS architecture and vision. In Industrial Informatics (INDIN 2009). 7th IEEE Int. Conf.

[9] http://www.zedboard.org

[10] http://www.xillybus.com

[11] http://www.nvidia.com/object/cuda_home_new.htm