A novel Embedded system for vision tracking

Antonis Nikitakis
Technical University of Crete,
Department of Electronic and Computer Engineering
Kounoupidiana, Chania, Crete, GR73100, Greece
anikita@mhl.tuc.gr

Theofilos Paganos
Technical University of Crete,
Department of Electronic and Computer Engineering
Kounoupidiana, Chania, Crete, GR73100, Greece
pagantheo@gmail.com@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. The OpenTLD algorithm uses a random forest classification method while it is very robust; however it cannot be efficiently parallelized in its native form. This is mainly due to the fact that it is a memory bound problem and 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 OpenTLD 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. Based on our real-world measurements our embedded system is more than 23 times faster than a state-of-the art Intel CPU and even faster that a highly parallel Graphical Processing Unit (GPU) while it is at least 40x more energy efficient than both the Intel CPU and the GPU. At the same time our system is highly scalable and can be seamlessly adapted for different computer vision schemes that are utilizing the Random Forest structures.

***Index Terms*— Embedded System, Object Tracking, Distributed Memory, Random Forests, Classification, FPGA.**

1. Introduction

The problem of long term visual tracking is very important in numerous application domains including surveillance, security, augmented reality and multimedia. The tracking process can be very difficult when the objects are moving fast, relative to the frame rate, and the object has to be tracked for a long period. “Long-term” object tracking refers to such 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 two most critical points in all those applications are the accuracy of the system as well as its real time performance at high resolutions.

All those tracking issues are efficiently addressed by the very promising and newly introduced OpenTLD algorithm [5]; OpenTLD is based on the Random Forest classifier introduced by Breiman Leo [1]. Even though the Random Forest classification scheme is being widely used in computer vision, it has a certain disadvantage over similar classification approaches; it cannot be efficiently parallelized at a fine grain (i.e. at the Random Forest access level).

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 state-of-the-art FPGA which incorporates a dual core ARM CPU, can execute this high-end detection 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; this speedup comes mainly from the utilization of our novel memory sub-system. At the same time our embedded system is faster even from a high-end GPU, while it consumes two orders of magnitude less energy than the conventional CPUs and GPUs. Moreover, the proposed memory scheme is modular, flexible, easily expandable and field-customizable.

2. 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 some random forest classifiers: they access different forests simultaneously. They utilize one classifier containing ten decision trees; all trees are processing the same input. They actually use multiple copies of the same image data (within a sub-window) for each decision tree. They get a 5x speedup when comparing their hardware performance with that of their own custom software implementation executed on a single core Intel CPU.

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 that the existing hardware approaches.

3. THE OPENTLD ALGORITHM

The authors of [5] investigate the problem of robust, long-term visual tracking of unknown objects in unconstrained environments. They propose a new approach, called Tracking-Modeling-Detection (TMD) that closely integrates adaptive tracking with online learning of the object-specific detector. In their object detector implementation, they use a sequential randomized forest classifier. The forest consists of several trees, where each of them is built from a certain group of features. Every feature in the group represents a measurement taken at a certain level of the tree. The extracted features are randomly partitioned into several same-size groups. Each group represents a different view of the patch appearance.

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 shown in the following simplified snippet the features identified are sampled in a randomized pattern in terms of position, scale and aspect ratio:

```
int feature_left = frame->Integral_image(x, y, w, h * 2);
        where:
x= (maxScale - minScale) * ((float)rand() / (float)RAND_MAX) + minScale;
y=…..
```

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++.

In the following sections we describe in detail the organization of our hardware scheme which is successfully prototyped on a modern embedded FPGA device.

4. HARDWARE ARCHITECTURE

Most of the hardware accelerators, in computer vision tasks, try to efficiently un-roll a big loop and then, by using a memory interleaving scheme, try to exploit the possible inherent parallelism. The openTLD [5] isn't any different but the memory access pattern, due to the use of the Random Forests' classifier, is random and can be badly distributed over the memory address space. Even a highly distributed memory architecture could not provide the necessary memory bandwidth without significant data repetition, as almost all the parallel memory queries triggered by the parallel processing cores are very often referring to the same memory block.

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 2 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.

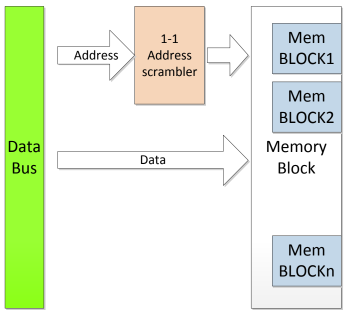

Figure 2. Address scrambling process, assuming a Data Bus loading scenario

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. In order for this scheme to work properly, two issues have to be solved. Firstly and more importantly, the scrambling process should be properly selected so as to maximize the parallel accesses. In the Random Forest classifiers, we realized that the accesses were performed in a randomized pattern within a sliding window, which had certain characteristics though: The accesses were bounded, in the vast majority of the cases (during a complete run) ,within a distance equal to the size of a memory block (1/32 of the image or 9600 pixels). Keeping this in mind we tried to redistribute the pixels contained in a single memory block to a much larger address space. Therefore, by inversing the order of the address bits, which is a 1-1 function with almost zero latency when implemented in hardware, we heavily decreased the memory collisions. This characteristic of the memory access pattern was found in all the Random Forest based applications we have investigated. Secondly, in order for our system to preserve the system's consistency, when more than one parallel queries access the same block (i.e. we have a collision) the queries are serialized. Figure 3 demonstrates our approach.

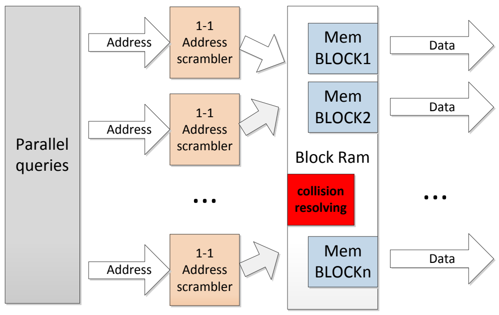

Figure 3. Random parallel queries referring on different block rams

Moving to the high-level architecture of our classifier, which utilizes our novel memory scheme, it comprises of three basic modules, the **Loop Decoding module**, the **Memory module** and the **Computation module** as shown in Figure 4.

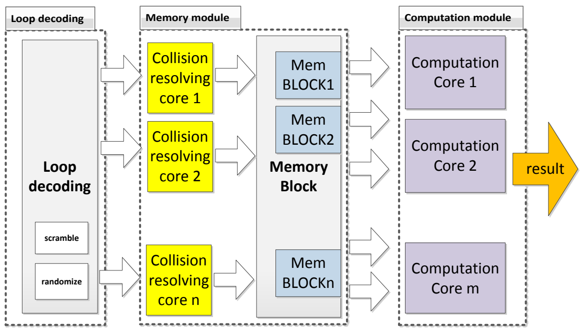

Figure 4. High level architecture

The **loop decoding** module implements the algorithm-specific tasks; in our case it is responsible for utilizing the sliding window movement, the tree traversing, the scale change while it is also responsible for the randomized sampling if/when applied to a certain window. It actually performs the loop-unroll while its output is a number of addresses corresponding to the different parallel queries. In our configuration we produce 16 queries per loop iteration.

The **memory module** comprises of all the memory blocks that form the classification memory of our system and contain the integral image. Each memory block is connected with a collision resolving core responsible for serializing the memory accesses to this specific block, in case of collisions. We explain that in detail in subsection B

The **computation module** implements the feature computation function; in our case it just calculates the function A-B-C+D while performing a simple memory query to a certain likehood table.

A. Loop decoding module

In this module we actually decode the main loop of the openTLD scheme which accesses the random forests. In particular, we implement the sliding window movement function as well as the randomized sampling method which uses the predefined coefficient matrices. Figure 5 shows an overview of this module.

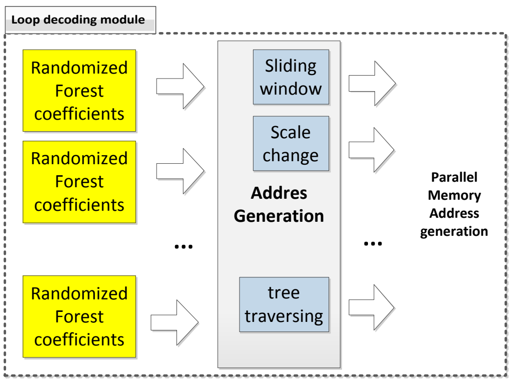
Loop decoding module
Randomized Forest coefficients
Randomized Forest coefficients
...
Randomized Forest coefficients
Sliding window
Scale change
Addres Generation
tree traversing
...
Parallel Memory Address generation

Figure 5. Loop decoding module

The loop decoding module produces 16 memory accesses per cycle after decoding 5 continuous iterations of the inner loop of the original openTLD code. It was primarily implemented with an 8-state Finite State Machine combined with 4 lookup tables (i.e. where the randomized coefficients are stored) which are connected with 4 fixed-point multipliers. In order to increase the performance we used 4 hardwired fixed point multipliers provided by the FPGA vendor. These multipliers can execute 1 computation per clock cycle with a tunable pipeline depending on the desirable clock rate.

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 very 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.

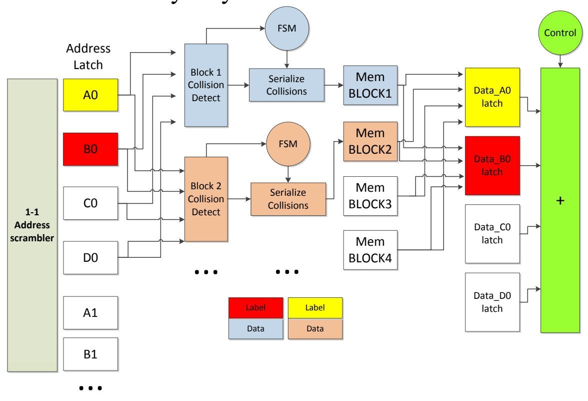

Figure 6. Memory module microarchitecture

Figure 6 demonstrates the dedicated collision resolving module which is attached to each block and which processes all the queries' addresses 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 the figures) 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 6), since this can be performed out-of-order.

C. The Computation module

The computation module is taking the data output of the Memory module and performs the actual classification 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. After the application specific computation is executed the necessary, in every Random Forest classifier, likelihood table memory lookup is performed in order to determine if the examined object is part of the trained dataset or not. Figure 7 demonstrates the architecture of this module.

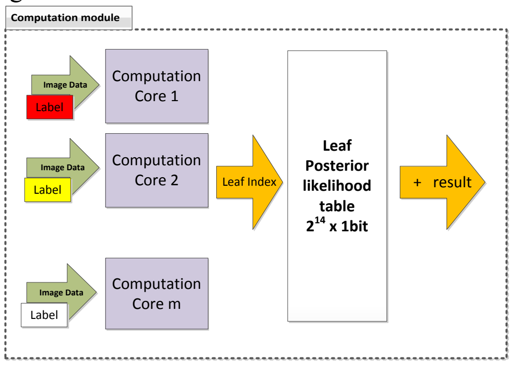

Figure 7. Computation Module

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.

5. Distributed Memory Analysis

Initially and in order to verify our novel approach we have utilized it in a real-world experiment. In particular, in order to get the memory access pattern to the classification object (i.e. the integral image) we intercepted the openTLD's execution in software and recorded its memory behavior. The inputs were real frames from a webcam and initially we have not altered/optimized the algorithm in any way. We decomposed all the nested loops and using the original random function we stored all the memory accesses in testbench files. We then recorded the memory patterns from our hardware system using the same input and the same values for the random function and we concluded that our memory sub-system handles correctly all the memory accesses.

We also used those real-world memory access patterns for measuring the performance of our system. In particular, we have carefully studied 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.

After experimenting with several hashing/scrambling functions we realized that the optimal one just inverses the order of the address bits; all the others, like crc16 and crc32, were producing similar performance results whereas they were more complicated in terms of hardware implementation.

In particular, reordering the address bits in hardware is a simple and zero latency function. The selected function redistributed the previously neighboring addresses into different blocks with a high probability as proved by our experimental results.

Table 1. Collision rate statistical analysis

Memory Distribution in blocks	Average collisions without scrambling[1]	Average collisions[1] 1-1 scrambling	Memory access[2] per cycle (dual port)
8	7.99	2.65	6.06
16	15.98	3.15	10.16
32	31.5	**3.6**	**17.78**

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. Those results clearly demonstrate that, if a conventional memory distribution scheme is used, increasing the number of blocks does not result in any increase in the performance since virtually all 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 32 memory accesses (i.e. by almost an order of magnitude) 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.

6. EVALUATION AND PERFORMANCE RESULTS

A. *Device utilization*

The system described in Section 6 has been initially prototyped in a Xilinx Virtex-6 VLX130T device. The utilization of the device is demonstrated in Table 2 and the critical resource was the number of the Memory Blocks, as expected. We selected this device, since in our case study, we could load on-chip a full 640x480 frame, in the integral format which uses 27-bits pixel depth.

Table 2. Hardware Cost on a Virtex-6 VLX130T device

Logic Utilization	Used	Utilization
Number of Slice Registers	1736/160000	1%
Number of Slice LUTs	6801/80000	8%
Number of fully used LUT-FF pairs	1307/7230	18%
Number of Block RAM/FIFO	237/264	89%
Number of BUFG/BUFGCTRLs	1/32	3%
Number of DSP48E1s	4/480	0%

B. *Performance Results*

In order to evaluate the performance of the proposed scheme we have executed the same application in a state-of-the-art CPU 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 activated.

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 and Figure 8 clearly highlight 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 section 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 device at 200Mhz

Memory partitioning in blocks (dual port)	Average collisions (with scrambling)	Speedup @ 200 Mhz vs Single Core CPU	Effective Memory BW (TB/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**

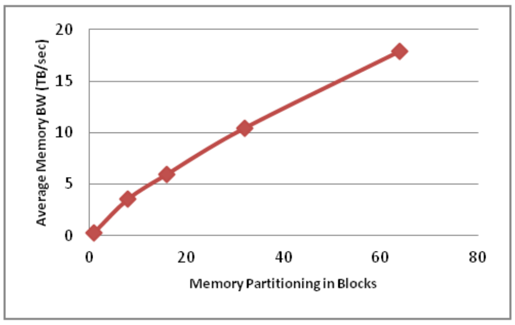

Figure 8. Average effective Memory BW vs Number of Memory Blocks[3]

To further prove that the speedup triggered by our novel system is very high even when compared with a modern multi-core CPU utilizing a dual channel memory, we implemented an openMP version of the original algorithm and utilized both cores of the 2.4GHz Intel CPU. The measured speedup, against the two-core implementation, triggered by

[1] average collisions per 32 memory accesses
[2]in average after a series of real-wolrd experiments

[3] We have implemented and tested up-to 32 dual port memory blocks

our FPGA approach when 32 block memories were utilized, is 26.16; 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 10TB/sec average measured bandwidth that our distributed memory system is providing.

C. *Embedded design and communication cost*

The recently introduced Zynq-7000 FPGA family from Xilinx allows for the implementation of real-world embedded systems exploiting both the performance of an FPGA accelerator as well as the flexibility of a modern dual core low power ARM CPU in the same die.

In the implementation of the complete embedded system (the Virtex-6 implementation of the last sub-section 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).

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 the 12.04 Ubuntu 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]. Xillybus offers several end-to-end stream pipes for data movements between the host CPU and the hardware resources at very high data rates. As all the stream pipes are buffered using dedicated FIFOs we are able to separate the two different clock domains of the bus and our hardware modules.

For our experiments we set the bus clock at 100MHz while our hardware scheme works at 200MHz. The Xillybus core supports higher bus clock rates, but due to some limitations on the DMA controller, for the specific ARM processor utilized, the use of a higher bus clock rate proved to be useless.

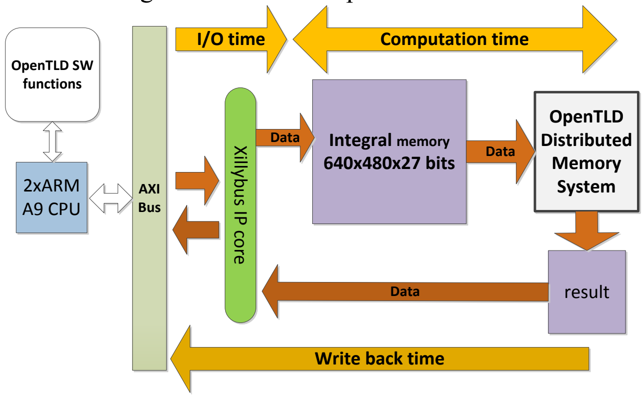

Figure 10. Xilinx Zynq-7000 verification scheme

In our proposed architecture, demonstrated in Figure 10, we transfer data from the CPU to the hardware modules practically only when we load the Integral image memory; then our hardware modules process those data until the complete image is fully processed.

In order to minimize the inter-communication overhead in this embedded application we chose to suppress the amount of data we had to move to the hardware side. This was possible by generating the integral image in hardware 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.

It is known [4] that the integral image can be computed efficiently in a single pass given a grayscale image i(x,y) using the formula:

$$I(x,y) = i(x,y) + I(x-1,y) + I(x,y-1) - I(x-1,y-1),$$

where I(x,y) is the Integral image and i(x,y) is the grayscale image.

As Figure 11 shows it is possible to compute on the fly the integral image data as we transfer the pixel data to the hardware modules by caching only 2 integral image lines (the current and the previous one). After each integral image pixel is computed it is redistributed to the memory blocks using our novel address scrambling scheme described in Section 5. The aforementioned approach can be easily synchronized with the streaming data coming to our hardware sub-system as the Xillybus core fully decouples the different clock domains providing synchronous read/writes from the Host to the hardware modules and vice versa.

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 device used.

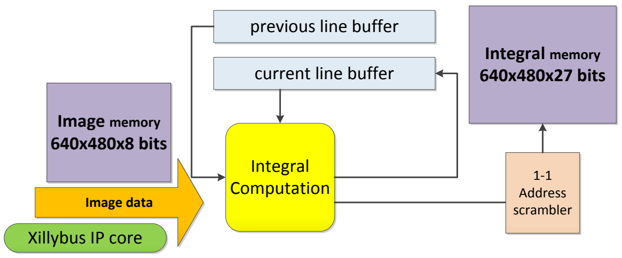

Figure 11. Integral computation scheme

Based on those measurements we need 1.46msec for the loading of an 640x480 image[4] 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.

The intercommunication time is dominated by the transfer of the image since the result, which should be sent back to the CPU, is a single 32bits datum and this is only needed once for each complete image. The update of the Leaf posterior likelihood array does not also trigger any significant overhead in the communication as only a few hundred values have to be updated between frames and this can be done fully in parallel with the actual processing.

Even though we may have a performance decrease because of the communication overhead (i.e. if the on-chip memory size does not allow for double buffering) our autonomous embedded system has still very high performance which can be translated to either more fps or to support, the very important today, multiple-object tracking. In this latter configuration the hardware processing time will be increased while the communication time will stay stable and thus we will minimize further the performance loss due to the data transfers.

D. 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 of the detect() function which is executed on the GPU device. The high level architecture of the GPU implementation is depicted in figure 12.

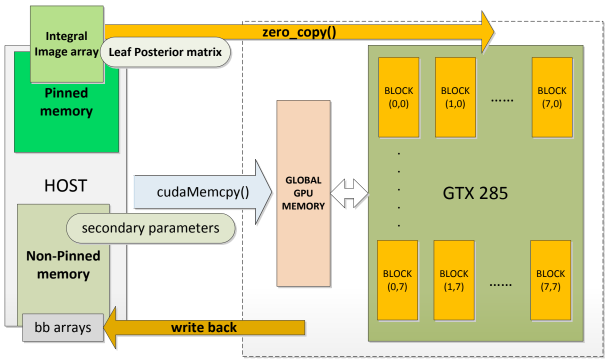

Figure 12: High level architecture of the Detector Module

The GPU utilized is the NVidia GTX 285 device and it is programmed using the well established CUDA API [11]. The kernel module we built in order to perform the detection phase needs basically the **Integral image** (640x480x32 bits) array and the **Leaf Posterior matrix** (2^{14} bits) which are computed in exactly the same way as in our embedded system. After the completion of the processing of each frame the host has to update these two arrays. We optimized our design using various techniques such as asynchronous memory copies and the zero copy memory optimization of CUDA in order to provide a non-blocking communication. The above setup activates a total number of 1024 kernels (i.e. threads).

By testing several video samples and setting training objects of different shapes and sizes, we confirmed a considerable acceleration to all those experiments when compared with the single-core CPU performance. In Table 4 we see 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 time (msec)**	**CUDA time (msec)**	**Speedup vs single-core**	**Embedded[5] time (msec)**	**Speedup 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

As those results clearly demonstrate our embedded system outperforms the GPU by at least a 2x factor when the I/O overhead is also taken into account while this speedup will be much higher when the DMA controller problem of 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.

E. 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. The only system that partially implements such a classifier is the one in [2]. However, in order to parallelize the system they have used a different approach: they are implementing, on the same device, more than one classifiers each running on a different input. Thus they are based on caching the image data for the parallel classifiers which takes advantage of specific characteristics of the application, poses many restrictions

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. Even when looking at the actual performance triggered by the application-specific approach of [2], it is not possible to have a direct comparison in terms of speedup with our system as they have not implemented a publicly announced openTLD clone but instead their own proprietary one.

In general, though, our system triggers a speed-up of more than 40x in comparison with a single core implementation whereas they claim a speedup of just 5 xwhen using their own software as a reference running on a much slower CPU. More importantly, our approach has certain advantages when compared with the one in [2]:

— Our system can support any combination of forest trees and classification features without changing a single wire in our hardware scheme. We have implemented up to 15 trees while supporting more than 12 features and our performance remained constant as the statistical characteristics of memory accesses have not been affected. In [2] they can only support 10 trees and 10 features due to hardware restrictions.

— Our system architecture is not setting any restriction in the sub-window size. In [2] the sub-window size is set to 1024 pixels.

— In their system the more processing cores they utilize the more memory blocks they need, as they are used as local caches. 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[6] as shown in Figure 9.

7. 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 20x 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

[6] under the assumption that the clock speed remains constant

numerous multimedia applications which involve the Random Forest approach.

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. *International Conference 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 Proceedings 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 International Conference on , vol., no., pp.319-324, 23-26 June .

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

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

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