Online ISSN 1848-3380, Print ISSN 0005-1144
ATKAFF 55(1), 9–21(2014)

Tomislav Matić, Ivan Aleksi, Željko Hocenski

CPU, GPU and FPGA Implementations of MALD: Ceramic Tile Surface Defects Detection Algorithm

DOI 10.7305/automatika.2014.01.317
UDK 681.532.2.015.42-13:691.4; 004.384.021
IFAC 2.8; 1.1; 5.6.7

Original scientific paper

This paper addresses adjustments, implementation and performance comparison of the Moving Average with Local Difference (MALD) method for ceramic tile surface defects detection. Ceramic tile production process is completely autonomous, except the final stage where human eye is required for defects detection. Recent computational platform development and advances in machine vision provides us with several options for MALD algorithm implementation. In order to exploit the shortest execution time for ceramic tile production process, the MALD method is implemented on three different platforms: CPU, GPU and FPGA, and it is implemented on each platform in at least two ways. Implementations are done in MATLAB's MEX/C++, C++, CUDA/C++, VHDL and Assembly programming languages. Execution times are measured and compared for different algorithms and their implementations on different computational platforms.

Key words: CUDA, FPGA, GPU, Integral Image, MALD, Ceramic Tile

CPU, GPU i FPGA implementacija MALD algoritma za otkrivanje nepravilnosti na površini keramičkih pločica. U ovom radu razmatra se prilagodba, implementacija i usporedba performansi metode pomičnog usrednjavanja s lokalnom diferencijom (MALD) s primjenom u otkrivanju površinskih nedostataka na keramičkim pločicama. Proizvodna linija keramičkih pločica je autonomna sve do zadnje faze u kojoj je potreban ljudski vid kako bi se otkrili eventualni nedostaci na keramičkim pločicama. Nedavnim razvojem računalnih platformi i razvojem metoda računalnog vida omogućena je implementacija MALD metode na nekoliko načina. U nastojanju skraćenja vremena potrebnog za proizvodnju keramičkih pločica, MALD metoda je implementirana u trima različitim platformama: CPU (central processing unit), GPU (graphic processing unit) i FPGA (field programmable gate array), te s barem dva različita algoritma. Implementacija je izvršena sa MATLAB MEX/C++, C++, CUDA/C++, VHDL te Asembler programskim jezicima. Izmjerena vremena obrade su međusobno uspoređena za različite algoritme i njihove implementacije na različitim računalnim platformama.

Ključne riječi: CUDA, FPGA, GPU, integralna slika, MALD, keramičke pločice

1 INTRODUCTION

Driven by the ceramic tile industry demand for faster production almost all of the production stages in the process are automated. The last but not the least important stage for quality control and classification is the bottleneck in the ceramic tile production process concerning automation [1]. Automation of this stage leads to machine vision systems with one or more digital cameras and one or more image processing algorithms [2], [3], [4]. We use a real-time ceramic tile production line equipped with machine vision for testing purposes, c.f. Fig. 1. Different algorithms are implemented depending on ceramic tile features. Algorithms can inspect chromatic abnormality, edge and corner defects, dot and blob shape defects, cracks, scratches, printed texture anomaly and glazing defects. Before applying above stated algorithms, most of the machine vision systems require some kind of image preprocessing. This can include a number of different algorithms for image alignment, rotation, segmentation and pixel brightness equalization. With human eye, the total ceramic tile failure detection time delay goes from 0.5 to 1 s [5]. From the economical point of view, shorter failure detection time leads to greater production and income. Capturing ceramic tile image with digital camera can be considered as a constant time delay process. Subsequently, total execution time can be reduced only by shortening the execution times of required machine vision algorithms. Better algorithm performance can be achieved by optimizing the algorithm code, by implementing a different method or by changing the computational platform used for the implementation and

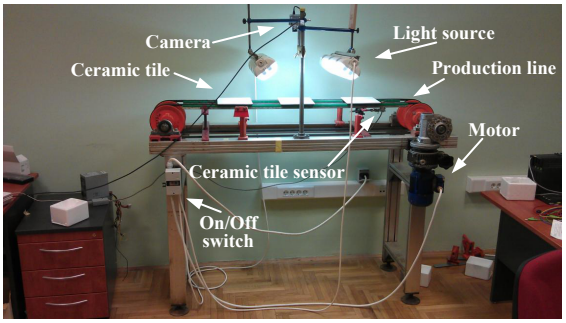


Fig. 1. Real-time ceramic tile production line equipped with machine vision system that is used for testing purposes in our laboratory.

execution. For this paper, three MALD algorithm implementations are considered, with and without integral image approach. They are implemented on different computational platforms: CPU, GPU and FPGA, with a code written in MATLAB's MEX/C++, C++, CUDA/C++, VHDL and Assembly programming languages.

Rest of the paper is organized as follows. Section II explains the MALD algorithm, integral images and MALD algorithm modifications. Sections III gives detailed explanation on three specific implementations of the MALD algorithm and how it is mapped to three different computational platforms. Experimental results are discussed in Section IV with conclusion in Section V.

2 MALD METHOD WITH INTEGRAL IMAGE APPROACH

Moving average with local difference method finds dot shaped defects on ceramic tile surface image [6], [7], [8]. MALD considers a symmetric 1D kernel, c.f. Fig. 2a, that can determine if ceramic tile image line is on the edge or not, and if the kernel center pixel can be considered as a defected pixel or not. The method takes 8-bit grey scale image I, with width X and height Y, c.f. Fig. 2b. Method also takes 4 input parameters: 1D kernel window width w, distance d as a number of pixels between the kernel window and the kernel center pixel $I(x,y)$, threshold pixel intensities t_1 and t_2 that define a ceramic tile edge condition (t_1) and defected pixel intensity (t_2). As an output, MALD returns a binary image B, with same size as I, where binary 1 corresponds to a defected pixel and binary 0 correspond to a non-defect pixel intensity. In Fig. 2a parameter d = 2 and parameter w = 3 are used just for illustration, while different values are used for implementation. Let $e(x,y)$ be the absolute mean value difference of the left and the right kernel side (1), and let $h(x,y)$ be the difference of the mean

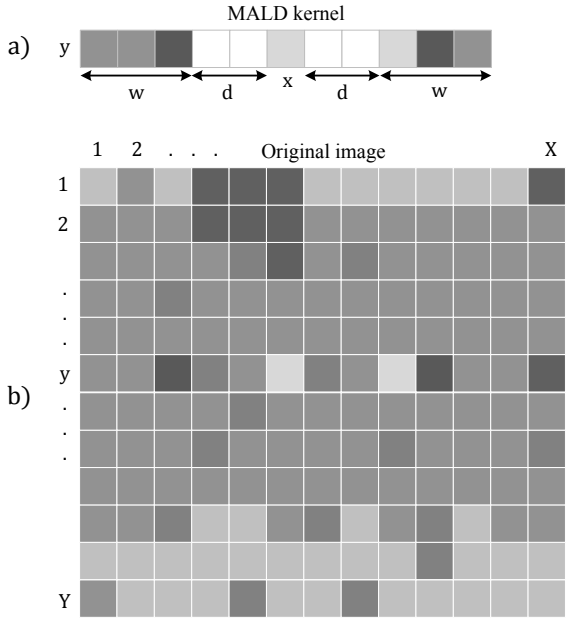


Fig. 2. Symmetric MALD kernel with d = 2 and w = 3 (a) with line y from; Original image (b).

values of both kernel sides and the pixel $I(x,y)$ (2).

$$e\,(x,y) = \frac{1}{w}\left|\sum_{i=x-w-d-1}^{x-d-1} I(i,y) - \sum_{i=x+d}^{x+w+d} I(i,y)\right| \quad (1)$$

$$h\,(x,y) = \frac{1}{2\,w}\left(\sum_{i=x-w-d-1}^{x-d-1} I(i,y) + \sum_{i=x+d}^{x+w+d} I(i,y)\right) - I(x,y) \quad (2)$$

MALD test for the edge condition compares $e(x,y)$ with t_1. If $e(x,y) < t_1$, the pixel $I(x,y)$ is considered located far from the edge and inside an image. Therefore it is analyzed further on. Otherwise, pixel $I(x,y)$ is considered as a pixel close to the edge and is skipped from further analysis, while the output $B(x,y) = 0$. The above condition avoids the edges and the environmental background outside a ceramic tile image.

Next stage of the MALD method determines if a pixel $I(x,y)$ intensity correspond to a defect by comparing $h(x,y)$ with t_2. If $h(x,y) \geq t_2$, pixel $I(x,y)$ is considered as defected pixel and the output binary image $B(x,y)$ is 1, otherwise 0. Finally, output binary image B gives the dot defect locations.

Input parameters for MALD method are different for different type of ceramic tile (one color, lightly colored, lightly textured). The best way to determine parameter

values is to execute the MALD algorithm on ceramic tiles without defects. Segmentation, rotation and alignment are not required since MALD method avoids the edges and the surrounding ceramic tile image background. If ceramic tile image is taken in uniform lightning conditions there is no need for pixel brightness equalization. When there is no need for the mentioned preprocessing algorithms, the total execution time can be decreased.

2.1 1D and 2D Integral Images

Integral image is an intermediate image representation that enables efficient calculation (in constant time) of sum of pixels that are inside a rectangular region [9] and is in a close relation to summed area tables in graphics [10]. Integral image II_{1D} at location $II_{1D}(x,y)$ is calculated from the original image I as the sum of all pixels to the left of $I(x,y)$. Analogously, integral image II_{2D} at location $II_{1D}(x,y)$ is calculated from the original image I as the sum of all pixels to the left and to the top of $I(x,y)$

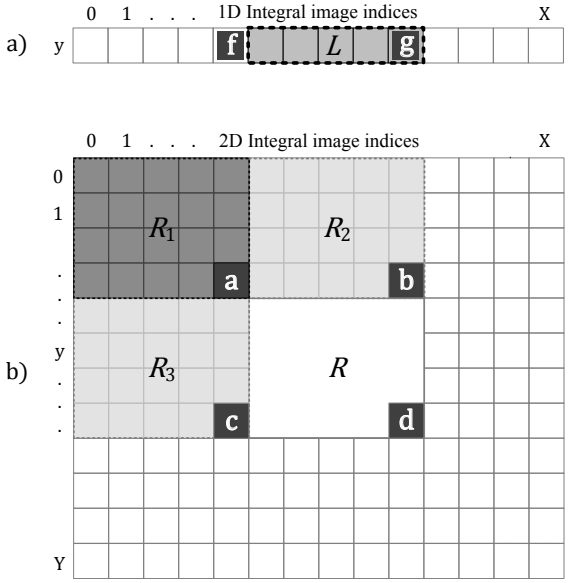


Fig. 3. Constant time sum of pixels in a rectangular region R by using integral image approach.

$$\begin{aligned} II_{1\mathrm{D}}(x,y) &= \sum_{u\le x} I(u,v) \\ II_{2\mathrm{D}}(x,y) &= \sum_{u\le x,\ v\le y} I(u,v) \end{aligned} \tag{3}$$

Once calculated, the integral image (3) enables calculation of a sum of pixels in constant time. With II_{1D}, the sum s_L of pixels inside a line region L is (4), c.f. Fig. 3a. With II_{2D}, the sum s_R of pixels inside a rectangular region R is (5).

$$s_{\mathrm{L}} = II_{1\mathrm{D}}(\mathbf{g}) - II_{1\mathrm{D}}(\mathbf{f}) \tag{4}$$

$$s_{\mathrm{R}} = II_{2\mathrm{D}}(\mathbf{d}) - II_{2\mathrm{D}}(\mathbf{c}) - II_{2\mathrm{D}}(\mathbf{b}) + II_{2\mathrm{D}}(\mathbf{a}) \tag{5}$$

where **a**, **b**, **c**, **d**, **f** and **g** are 2D indices (x,y), c.f. Fig. 3. With 1D integral image, values at indices **f** and **g** have assigned a cumulative row sum (3) of all pixels from index $(0,y)$ to (x_f,y), and from index $(0,y)$ to (xg,y), respectively. Therefore, the sum s_L (4) is a difference of cumulative summations $II_{1D}(\mathbf{g})$ and $II_{1D}(\mathbf{f})$. Analogously, with 2D integral image II_{2D} (3), the sum s_R (5) is the sum of pixels from regions $\{R, R_1, R_2, R_3\}$ with subtracted summations from regions $\{R_1, R_3\}$ and $\{R_1, R2\}$. Since the summation from the region R_1 is subtracted twice, it has to be added once, c.f. Fig. 3b. Finally, the benefit from summing image pixels with integral image comes from reduced number of memory accesses and less ALU operations used. Summation s_L has only 2 memory accesses and 1 addition operation, while s_R has 4 memory accesses and 3 ALU operations. Integral image can be calculated with zero-based indexing as with Alg. 1.

Algorithm 1 Integral image calculation from original image

```
1.  Create array II_1D with size [X+1, Y], and create array II_2D
    with size [X+1, Y+1], both initialized to zero.
2.  for y = 1 to y = Y
3.      for x = 1 to x = X
4.          II_1D(x,y) = II_1D(x–1,y) + I(x,y)
5.  for x = 1 to x = X
6.      for y = 1 to y = Y
7.          II_2D(x,y) = II_2D(x,y–1) + II_1D(x,y)
```

Cumulative row sum for calculation of II_{1D} is done with Alg. 1 (*lines* 2-4). After Alg. 1 (*lines* 5-7), II_{2D} is created as a cumulative column sum of II_{1D}. Integral image II_{1D} has one more column than the original image, while 2D integral image has one row more than 1D integral image.

2.2 MALD with 1D Integral Image Approach

MALD method uses 1D kernel, c.f. Fig. 2a, and most of MALD's execution time is spent on the mean value calculation, for each line independently, eq. (1) and (2). 1D integral image approach is involved in order to calculate the mean value faster, c.f. Alg. 1. In order to get the mean value directly from the summations contained in II_{1D}, Alg. 1 (*line* 4) can be replaced with (6), where summations are divided with the number of pixels for the mean value calculation.

$$II'(x,y) \;=\; II'_{1\mathrm{D}}(x-1,y) \;+\; I(x,y)/w \tag{6}$$

To decrease the amount of time needed for mean value calculation, (1) and (2) can be calculated faster by using normalized 1D integral image *II'* (6). MALD kernel modifications are done in conjunction with *II'*. As depicted in Fig.

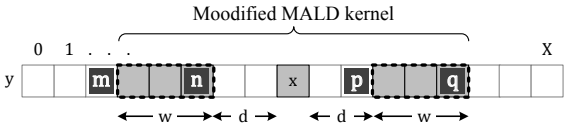


Fig. 4. MALD kernel modified in conjunction with (7) with only 4 indices required.

4, modified kernel has only 4 pixel indices for (1) and (2) calculation reformulated with *II'* in (7) and (8).

$$e(x,y) = II'(\mathbf{q}) - II'(\mathbf{p}) - II'(\mathbf{m}) - II'(\mathbf{n}) \quad (7)$$

$$h(x,y) = (II'(\mathbf{q}) - II'(\mathbf{p}) + II'(\mathbf{n}) - II'(\mathbf{m}))/2 - I(x,y) \quad (8)$$

3 IMPLEMENTATIONS

Three different implementation types of MALD are considered in this paper, c.f. Tab. 1, for ceramic tile image with size [*X*,*Y*] and the kernel parameters *w* and *d*. Total number of required ALU operations for calculating (1) and (2), per line *y* = 1, ..., *Y*, is presented in Tab. 2. First type $MALD_1$ is a straightforward approach which sums all pixels at each of N_x=*X*–2(*w*+*d*) kernel locations (*x*,*y*). $MALD_1$ has the highest number of ALU operations since it is starting with the sum equal to zero every time. Calculation of (1) uses 2*w* '+' and 1 '–' operation for N_x kernel locations. Second type $MALD_2$ uses current kernel pixel sum, at location (*x*,*y*), in order to calculate the sum of pixels at the next kernel location (*x*+1,*y*), as it is implemented in [6]. $MALD_2$ uses moderate number of ALU operations, c.f. Tab. 2. Next kernel sum is calculated from current kernel sum value by subtracting the first pixel intensity value in the current kernel window and by adding the last pixel intensity value in the next kernel window. Initially, it uses 2*w*+1 ALU operations. As kernel moves by single pixel, next kernel location sum (1) is calculated by single pixel intensity subtraction and addition for both kernel sides, respectively. Therefore, it uses 5 ALU operations for N_x–1 kernel locations. Third type $MALD_3$ uses integral image approach, and has the least number of calculations, c.f. Tab. 2. Integral image is calculated initially with *X* number of ALU operations per line. Further on, the modified MALD kernel (7) is applied, using 3 ALU operations per line. Finally, proposed MALD implementations are done for CPU, GPU and FPGA platforms, as it is illustrated in Tab. 3. Three MALD implementation types are compared with generic input parameters and are presented in Tab. 2. Division operations per image line for calculating (1) with $MALD_1$ and $MALD_2$ are equal to the number of kernel locations N_x. Number of divisions per image line with $MALD_3$ equals to image width *X* since normalized integral image is required to calculate for the whole line. Comparison between MALD implementations with practical input parameter values, c.f. Tab. 3.

Table 1. MALD implementation platforms.

Platform	$MALD_1$	$MALD_2$	$MALD_3$
CPU	✓	✓	✓
GPU	✓	–	✓
FPGA	✓	✓	–

Table 2. Number of ALU operations per image line that are executed when calculating (1).

MALD type	Number of ALU operations per line *y*	
	Additions (+) and subtractions (–)	*Divisions (/)*
$MALD_1$	$X-2(w+d)(2w+1)$	$X-2(w+d)$
$MALD_2$	$(2w+1)+5((X-2(w+d))-1)$	$X-2(w+d)$
$MALD_3$	$X+3(X-2(w+d))$	X

As depicted with Tab. 3, the algorithm $MALD_1$ has the highest number of ALU operations and it is considered as a worst-case scenario for sequential CPU implementation. Algorithm $MALD_2$ is not implemented in GPU, c.f. Tab. 1. It calculates the kernel sum recursively, where the next sum value depends on the previous kernel sum value. Therefore, $MALD_2$ is considered as inconvenient for GPU parallel implementation. On the other side, algorithm $MALD_3$ requires the whole image storage in memory, denoting more RAM resources, and calculation of (6), denoting more time and power to calculate. With FPGAs, integral image for MALD algorithm losses its purpose, as proposed in Section II. Due to high level of parallelism and efficient pipeline, FPGAs can calculate $MALD_1$ output in a pixel per clock cycle, with no need for integral image approach with $MALD_3$, c.f. Tab. 1.

3.1 CPU Implementation

In this paper is considered a sequential CPU implementation in Visual C++ and MATLAB's MEX/C++ programming environment. Implementation is done on a PC with Intel Core2 QUAD 6600 (one core used) with 6GB RAM and Windows 7 x64 OS. The algorithms are developed with Visual Studio 2010. MATLAB was used for MALD algorithm development and MEX/C++ and C++ for the implementation of all three MALD types, c.f. Tab. 1. Development of C++ code with MATLAB engine is presented in [11]. It provides with flexible implementation environment by using debugging functions which shortens

Table 3. MALD '+' and '-' operations/line for typical parameters (d = 5).

w	X	$MALD_1$	$MALD_2$	$MALD_3$
16	1024	32406	4943	3970
32	1024	61750	4815	3874
64	1024	114294	4559	3682
16	1536	49302	7503	6018
32	1536	95030	7375	5922
64	1536	180342	7119	5730
16	2048	66198	10063	8066
32	2048	128310	9935	7970
64	2048	246390	9679	7778

functional verification time. All three MALD types are implemented in C++ and MEX C++. MALD MEX function enables usage of MALD method in MATLAB with better performances, i.e. shorter execution time [11]. MEX functions are written in lower programming language, with approximately 125 times shorter execution time.

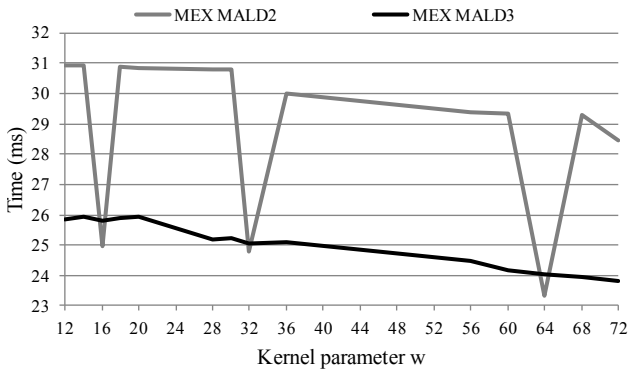


Fig. 5. Execution times of MALD MEX functions for varying kernel window size w.

As expected, sequential implementations have their execution times in accordance with the number of required operations, c.f. Tab. 2. Within a C++ program, the $MALD_1$ method has the largest, the $MALD_2$ has moderate, and the $MALD_3$ has the lowest execution time. However, MEX/C++ implementation of the $MALD_2$ method has shorter execution times than the $MALD_3$ only in cases when the kernel window length w is power of 2, as it is illustrated with the example in Fig. 5. Execution time slightly decreases with the increasing of the parameter w due to the less memory accesses to image data, since with the MALD kernel $2(w + d)$ pixels per line (left and right side edge pixels) are not considers. $MALD_1$ is not illustrated since its execution time is approximately 3 times greater than the two displayed in Fig. 5.

In order to achieve better performances, regarding execution times, VS 2010 C++ Compiler options are:

- Optimization → Full Optimization / Favor Fast Code / Whole Program Optimization,
- Code Generation → Floating Point Model → Fast.

Improvement of the execution time can also be achieved by setting the in the *release mode* and code compiling with a specific processor affinity and a real-time process in the Task Manager of the Windows operating system.

3.2 FPGA Implementation

Field Programmable Gate Array (FPGA) technology enables parallel algorithm implementation using Hardware Description Language (HDL). Regarding performances, FPGA's belong between a Central Processing Unit (CPU) and an Applicative Specific Integrated Circuits (ASIC). Generally, CPU solutions are known as flexible solutions, i.e. one processor can solve more than one real-world problem, while ASIC solutions are known as a solution with the best performances for solving a single real-world problem. Two types of CPU cores can be used on FPGA: a soft-core CPU that is made of FPGA's logic cells, and a hard-core CPU that uses part of integrated circuitry. For instance, FPGA soft-core RISC processor is an 8-bit PicoBlaze CPU with proprietary license limited to Xilinx devices only.

Real-time FPGA ceramic tile processing solution can be found in [12], where a machine vision system is equipped with a line-scan camera that records ceramic tile images in a line-per-line manner and a Finite State Machine (FSM) is used for image processing. However, in this paper we considered an FPGA-based approach with pipelined parallel $MALD_1$ and sequential $MALD_2$ implementations, proposed in Section II. Presented FPGA implementation uses DATA2MEM tool for downloading image lines to the Xilinx's Spartan3 XC3S200 development board. Pipelined parallel FPGA implementation is done in concurrent VHDL without an FSM, while sequential implementation is done in Assembly language for the PicoBlaze softcore CPU.

In order to parallelize the $MALD_1$ algorithm, in this work a 16-bit pipelined array adder is used that adds two values each time, c.f. Fig. 1. In the first stage array adder adds 1^{st} with 2^{nd} value, 3^{rd} with 4^{th} value, etc. In the next stage it adds the sum of 1st and 2nd with the sum of 3rd and 4th value, etc, c.f. Fig. 6. In order to calculate the sum of w inputs, for a single kernel side, the number of inputs is zero-padded up to the number of $2^{\lceil \log_2 w \rceil}$ 8-bit input values. Since MALD requires subtraction of mean

values of two kernel sides in (1), one subtraction is used for calculation of (1) and one addition is used for the calculation of (2). Left and the right kernel side sums are stored in a memory, from where (1) is calculated by subtraction, while (2) is calculated with addition. Array adder uses N_s (9) 16-bit unsigned integer full adders, where $\lceil . \rceil$ denotes ceil operation.

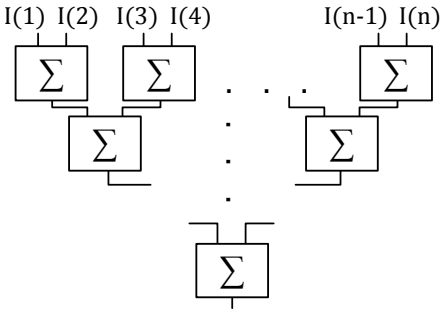


Fig. 6. Pipelined array adder stores a result of two element summations in a register which is used for next addition in next clock cycle.

Bottleneck to the parallel solution is a RAM access that enables single memory value read per clock cycle. In order to access required pixels instantaneously, a shift register with $2w + 2d + 1$ (*kernel size*) 8-bit elements is used as an input array. Shift register is initially filled with first *kernel size* number of pixels. As kernel moves through a line, first pixel is removed from the input array with single pixel shift operation (shift register) and the last pixel is inserted to the input array. In this work we used shift right operation to divide the sum with w for mean value calculation. Therefore $w = 32$ is chosen as the power of two and is a fixed number. It is also possible to use the Xilinx's CoreGen IP Core for division that is also done in one cycle. However, proposed pipelined parallel MALD implementation has no limitations to an FPGA manufacturer. Finally, pipelined parallel FPGA MALD outputs a 16-bit value of (1) and (2) which are calculated in a single clock cycle for each kernel location. Calculation is initially delayed for: *kernel size* memory read cycles; 5 cycles for propagating the first output value; 1 cycle for (1) subtraction or addition (2) of left and right kernel side mean values; 1 cycle for a division operation.

$$N_s = \sum_{k=0}^{\lceil \log_2 w \rceil - 1} 2^k \tag{9}$$

Sequential FPGA MALD_2 implementation is done with a softcore 8-bit PicoBlaze CPU. It is a simple 16-bit summation of 8-bit input data. The 16-bit summation was done with 8-bit commands for addition (ADD) of lower bytes and addition with carry (ADDCY) for addition of higher bytes. Subtraction is done analogously with SUB and SUBCY commands. Since RAM addresses are 16-bit, two 8-bit output registers are used for memory access, i.e. for reading pixel intensity values.

Used FPGA resources for the sequential and pipelined parallel implementations are illustrated in Fig. 7. Parallel implementation is faster than the sequential one, what is compensated with used more FPGA resources. About 6 times more FPGA slices and Flip Flops and about 2.5 times more LUT's are used. Block RAMs are used twice more for the sequential implementation since one BRAM is for the image storage, while the other is for the PicoBlaze's instruction RAM.

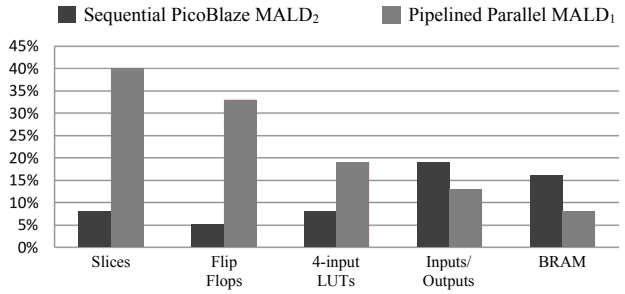


Fig. 7. Used FPGA resources for the sequential and the pipelined parallel MALD_1 implementation.

Execution time of both pipelined parallel and sequential implementation does not depend on pixel intensities. With change of kernel size, the number of elements in the array adder (9) changes proportionally, while the execution time remains the same for arbitrary kernel window size, c.f. Tab 4. However, the FPGA solutions are adjusted for the line-scan camera [9] and the results for the whole image are calculated approximately as if the whole image is written in FPGA RAM.

Total FPGA power consumption is the sum of design dependent *dynamic power* and constant hardware related *quiescent power*, as it is illustrated in Tab 4. Quiescent power is the sum of the *device static power* and the *design static power*, and it varies with temperature [13]. Dynamic power increases with frequency and operating temperature, and it varies by technology and architecture. Dynamic power consumption is related to implemented design and comes into the game when logic cells are switching. It mostly comes from glitches, caused by different gate delays, which results in multiple signal change during single clock cycle [13]. Dynamic power consumption at certain operating frequency is illustrated in Fig. 8, for two different MALD implementations. On one hand, sequential PicoBlaze implementation of MALD_2 has slightly decreasing dynamic power with newer FPGA family. On the

other hand, pipelined parallel MALD implementation has varying dynamic power. For the case of Spartan 3 family, one can conclude that the pipelining reduces the number of spurious glitches which, in turn, reduces dynamic power [14], [15]. However, with increasing frequency and varying FPGA technology, pipelined design consume more dynamic power in comparison with smaller sequential PicoBlaze design.

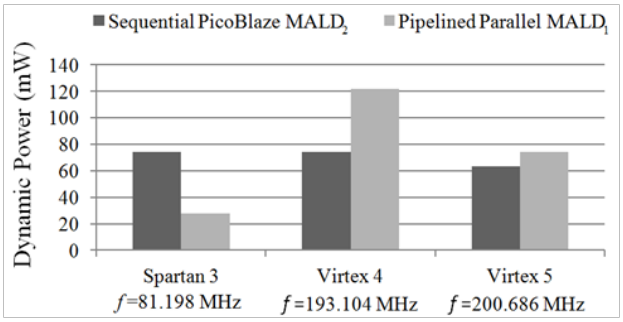


Fig. 8. FPGA's dynamic power consumption for two different MALD implementations.

3.3 GPU Implementation

Since recent CPUs are limited with the operating frequency, typically 3 GHz, parallel computing enables accelerated computation. Another available parallel computing platform is the graphic processing unit (GPU). At the present time, two major GPU vendors enabled the use of GPU for general purpose computing (GPGPU): NVIDIA and AMD [16]. GPGPU has a variety of applications in computationally demanding areas that require parallel processing of large data sets.

Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model developed by NVIDIA. It uses a GPU for general-purpose programming. GPU is a highly parallel machine with hundreds of processors that execute thousands of threads. CUDA is built around a scalable array of multithreaded Streaming Multiprocessors (SMs). Each multiprocessor consists of eight Scalar Processor (SP) cores, with two special function units for transcending a multithreaded instruction unit and an on-chip shared memory (NVIDIA's 8-series GPUs). CUDA extends C with functions called *GPU kernels* which can be executed on demand with *N* times in parallel by *N* different CUDA *threads*.

CUDA structures GPU kernels into parallel thread blocks. Programmers specify the number of thread blocks and threads per block, and the hardware and drivers map thread blocks to SMs on the GPU. Threads within a block can cooperate among themselves by sharing data through shared memory and synchronizing their execution to coordinate memory accesses [17]. Beside shared memory GPU also has read-write global memory, read only constant memory and read only texture memory accessible to all threads. These memory spaces are optimized for different memory usages. Texture memory is cached and the cache is optimized for 2D spatial locality, i.e. threads that read texture addresses that are close together will achieve better results [18].

Recently, CUDA technology has been used for image segmentation and classification [19, 20, 21, 22], 3D simulations [20, 21], different mathematical computations [25, 26], etc. Most of the GPGPU-based research is about the algorithm speed up and their real-time application. Authors in [19] achieved 10-50 times speedup for the GPU version of the Quick shift algorithm. Speedup factor of 360 was achieved in [20] for co-occurrence matrix calculation and Haralick texture features extraction. In [22] a real-time non-parametric color image segmentation method has been implemented on the GPU platform.

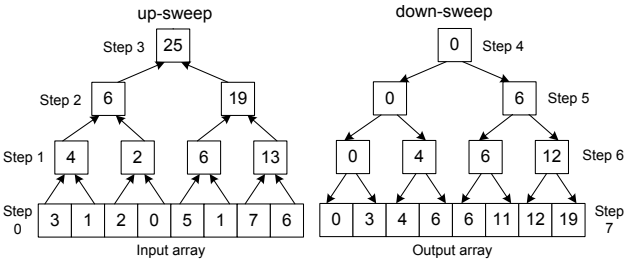


Fig. 9. Parallel prescan operation with tree representation of two phases.

According to Tab. 1, $MALD_1$ and $MALD_3$ methods are implemented in GPU with CUDA technology. CUDA implementation of the $MALD_3$ algorithm consists of two parts: parallelization of 1D integral image with the all-prefix-sum operation, and the GPU parallelization of MALD algorithm with II_{1D} approach ($MALD_3$). On the other hand, CUDA implementation of $MALD_1$ algorithm consists of only GPU parallelization of MALD algorithm.

CUDA utilizes the all-prefix-sum operation to calculate 1D integral image. The all-prefix-sums operation takes a binary associative operator $\oplus$ and an ordered set of n elements [$a0$, $a1$, ..., an-1], and returns an ordered set [a_0, $a_0 \oplus a_1$, ..., $a_0 \oplus a_1 \oplus \cdots \oplus a_{n-1}$].

In [21] the all-prefix-sum operation is used on a data segment. Similarly for this paper, the all-prefix-sum operation is executed on a single line (data segment) of the ceramic tile image where $\oplus$ is the addition operator. Output of the operation is an equally sized array in which each element value is the sum of all preceding elements, which

Table 4. Power consumption, maximum operating frequency and execution times for sequential and pipelined parallel MALD implementations on three FPGA families and ceramic tile image with 1024 pixels per line.

Method	FPGA	Power (mW)			Frequency (MHz)	Execution time (μs)
		Total	Dynamic	Quiescent		
Pipelined Parallel $MALD_1$	Spartan 3	69	28	41	81,198	20,0
	Virtex 4	287	122	165	193,104	8,4*
	Virtex 5	637	74	563	200,686	3,4*
Sequential PicoBlaze $MALD_2$	Spartan 3	115	74	41	126,961	3820
	Virtex 4	238	74	164	204,368	2373*
	Virtex 5	626	63	563	218,076	2224*

* *value is calculated relative to Spartan 3 measured execution time, according to estimated max. operating frequency.*

is the so-called *scan operation*. Beside the scan operation the all-prefix-sum can result in an array that contains the sum of all previous elements but without the final element, the so-called *prescan operation*. Prescan operation returns the ordered set [0, a_0, $a_0 \oplus a_1$, ..., $a_0 \oplus a_1 \oplus \cdots \oplus a_{n-2}$]. Prescan result can be generated from the scan result by right shift operation and zero padding from the left. Analogously, scan can be generated from the prescan result, respectively.

Although the all-prefix-sum is inherently sequential algorithm, there is an efficient parallel solution that consists of two parts: the up-sweep and the down-sweep. As illustrated in the example, c.f. Fig. 9, input requires an array length n to be a power of two. Each tree level in the up-sweep can be executed in parallel where the number of used processors is reduced with every step by two. Step 0 uses *n/2* processors, since every processor adds only two values. Step 1 uses *n/4* processors etc. Result of the up-sweep is the total sum of the array and the partial sums (step 1 and step 2). Partial sums are used in the second down-sweep phase, i.e. in the prescan. The down-sweep starts at the root of the up-sweep tree, with the root value replaced with zero. Each step of the down-sweep, the vertex value from the previous step is added to the left child and copied to the right child, while the left child value is set to the vertex value. Finally at step 7 resulting output array is the so-called prescan of the input array [27]. Development of the prescan algorithm is explained in [28] and implemented in the CUDA's data parallel primitives library (CUDPP). For 2D array it can perform prescan and scan operations on each row independently, without zero column padding. In this work, zero padding is implemented by checking if index value is –1.

CUDPP algorithms are executed using the algorithm interface functions and the plan interface functions. Plan Interface functions are used for creating CUDPP plan objects that contain configuration details and intermediate storage space. Algorithm interface functions execute specific algorithm based on configuration details in the plan object [29].

MALD algorithm is implemented using Alg. 2 with CUDA capable NVIDIA's 8-series GPU 9800 GT with 112 CUDA cores and compute capability 1.1 [30]. Input image I and the integral image II_1D are global read-only texture references which can be accessed inside the kernel using CUDA texture fetch functions. CUDPP plan is created in order to calculate integral image II_1D with the scan algorithm. After initialization, Alg. 2 (lines 5-10) executes a loop in which an input image is copied from CPU to GPU. The output array B is initialized to zero values and integral image is calculated by executing the scan operation plan. CUDA kernel, defined with Fig. 10, is executed and the output B is copied from a GPU to CPU. Finally, the CUDPP plan is destroyed and the global and the local GPU memories are freed.

Algorithm 2 Parallel MALD method on a CUDA GPU.

1. Find a CUDA capable GPU.
2. Allocate global GPU memory for I, $*II_{1D}$ and B.
3. Bind input data to a texture reference.
4. (*) Create CUDPP plan object for the *prescan* operation.
5. **while** next input image I is available **do**
6. Copy new image I to a GPU global memory.
7. Set B to all zeros in a GPU global memory.
8. (*) Execute the scan operation interface function for II_{1D}
9. Execute CUDA GPU kernel, c.f. Fig. 9.
10. Copy output binary image B to a CPU RAM memory.
11. Free GPU global memory.
12. Unbind textures.
13. (*) Destroy the CUDPP plan object.

(*) *lines used for $MALD_3$ method and not used for $MALD_1$ method.*

$MALD_3$ kernel is illustrated in Fig. 10 b. $MALD_3$ kernel sum is calculated N_x=X–2(w+d) times per line. CUDA kernel executes $N_x \cdot Y$ number of threads in parallel, for im-

age size [*X*, *Y*]. Since the number of required threads exceeds the limit of 512 threads per block, a grid of blocks *G*3 is organized as

$$G_3 = \left(\frac{(N_x + N_{tx} - 1)}{N_{tx}}, \frac{(Y + N_{ty} - 1)}{N_{ty}} \right) \qquad (10)$$

Where N_tx = 8 represents the number of threads in the *x* dimension of the block and N_ty = 8 represents the number of threads in the *y* block dimension. By knowing the number of blocks and the maximum number of threads per block (10) one can guarantee that the total number of threads is always equal or greater than the required $N_x \cdot Y$ number of threads. Maximum number of threads per block depends on the GPU series [18].

$MALD_1$ GPU implementation uses Alg. 2 without (*lines* 2*, 4, 8 *and* 13). In comparison with $MALD_3$, $MALD_1$ doesn't use integral image and CUDPP plan. $MALD_1$ kernel window sum is also calculated *N*x times per line and the GPU kernel executes *N*x·*Y* number of blocks (11).

$$G_1 = (N_x, Y) \qquad (11)$$

Each block has $2^{\lceil \log_2 w \rceil}$ threads, grouped as $(N_{tx}, N_{ty}) = (2^{\lceil \log_2 w \rceil}, 0)$. In each block, *w* threads are used for parallel data copying from a GPU global memory to a GPU shared memory, while the rest of the threads are used for setting the unused shared memory values to zero. Thread synchronization ensures that all data is copied in a shared memory before continuing with the next step, c.f. Fig. 9a. Parallel summation of data in shared memory is done with the reduction algorithm, c.f. Fig. 6. CUDA reduction algorithm [31] uses $2^{\lceil \log_2 w \rceil}$ threads for the both kernel window sums separately, c.f. Fig. 2a. Again, thread synchronization is used to ensure that the summations are finished and stored in the 0th element of the block shared memory buffers. In this way, separate summations of the left and the right side of the $MALD_1$ kernel window, c.f. Fig. 2a, are stored in a GPU shared memory. In order to avoid non-necessary memory accesses, the rest of the GPU kernel is executed by only one thread, c.f. Fig. 10a.

4 EXPERIMENTAL RESULTS

Experimental results are obtained with three computational platforms, as it is proposed in Section III. CPU and GPU implementations are considered with the following constraints. Ceramic tile images are considered with sizes [*X*,*Y*] ∈ {(1024, 1024), (1536, 1024), (2048, 1024), (2048, 2048)} Typical ceramic tile image is illustrated in Fig. 11a. Three kernel window sizes are used *w*∈{16, 32, 64}, c.f. Fig. 5, while other kernel

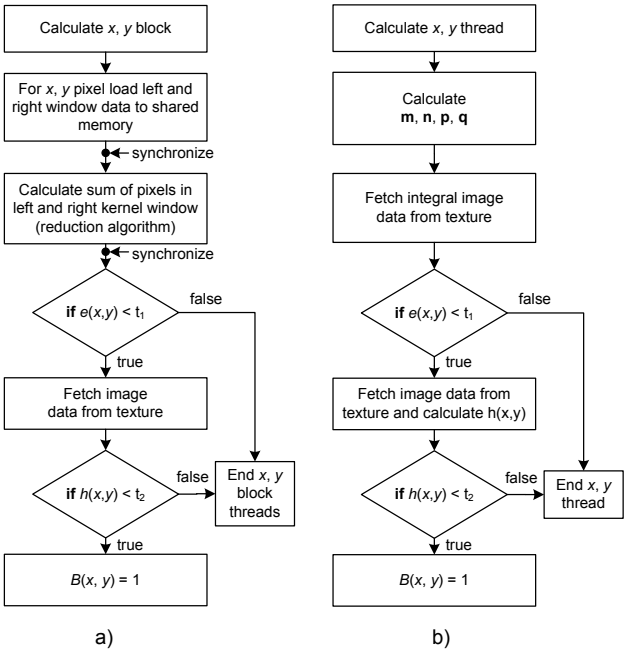
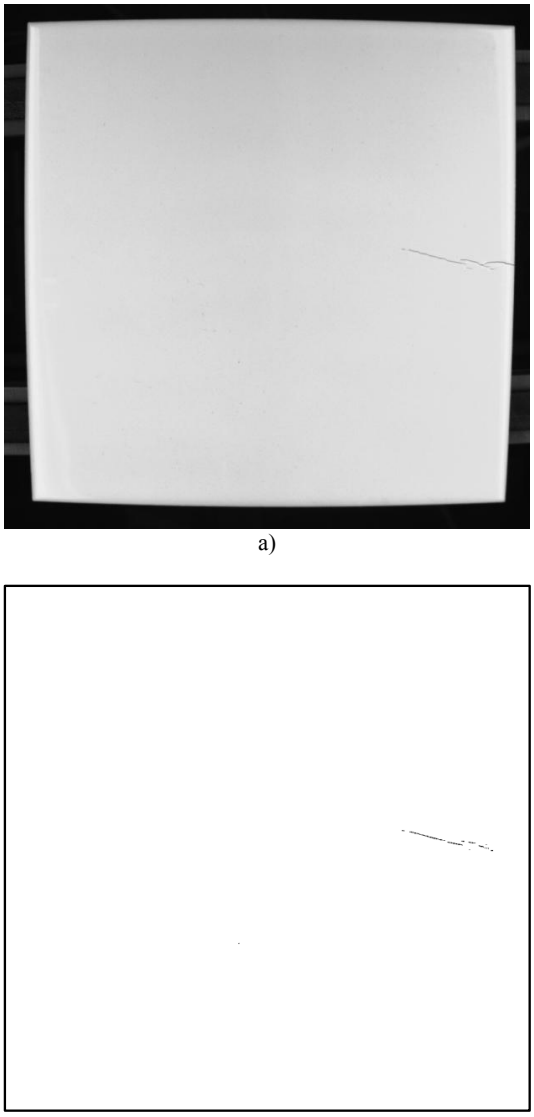


Fig. 10. GPU kernel of: $MALD_1$ (a); $MALD_3$ (b).

window parameters are fixed: *d* = 5, t_1 = 5 and t_2 = 30. Average execution times are calculated with 50 iterations. On the other hand, the FPGA implementations, presented in Tab. 1, have fixed execution times that are related to the maximum operating frequency of the MALD's hardware design and the properties of the target FPGA. Spartan-3, Virtex-4 and Virtex-5 are considered in this paper. Virtex-5 FPGA implementation of $MALD_1$ has the smallest execution time value of 1.8 μs per line with 1024 pixels. Therefore, the whole image with the fixed parameter *w* = 32 and height Y∈(1024, 1536, 2048), can be processed on the FPGA in (1.843, 2.765, 3.686) ms.

Execution times of MALD implementations, c.f. Tab. 1, for CPU, GPU and FPGA platforms are illustrated in Fig. 12. Ceramic tile image with the size of (1024, 1024) pixels is taken with the BASLER A102FC camera which is mounted on the ceramic tile production line, c.f. Fig. 1. MALD execution times, c.f. Fig. 12, are in accordance with the number of operations calculated in Tab. 3, with only one exception. MEX $MALD_2$ has better calculation times than MEX $MALD_3$ only in cases when the kernel window parameter *w* is the power of two, c.f. Fig. 5.

FPGA results are calculated from the execution time required to process a single line. Beside the shortest execution time, FPGAs have additional advantage due to the fact that the execution time has exact length. On the other hand, CPU and GPU are OS (Windows in this case) driven systems which can affect timings by calculating something



a)

b)

Fig. 11. Ceramic tile input image I with size (1024,1024) with defects (a) and the output binary image B with detected defects (b).

in the background, i.e. multitasking. Overall, GPU outperformed CPU in both, $MALD_1$ and $MALD_3$ implementations. However, GPU has better performances as image becomes larger, c.f. Fig. 13. MALD implementations are executed on variable image sizes, in order to illustrate that with larger input data, the parallel computation capability becomes more effective. Subsequently, for sequential algorithms, as it is the case with the 1D integral image algorithm, Alg. 1, CPU outperforms GPU, as depicted in Fig. 14. Since integral image creation is a sequential process, best performances are obtained with the CPU sequential implementation. Parallel GPU implementation with the all-prefix-sum is not as fast as it is complicated to implement. MATLAB's MEX/C++ functions execute the same calculation about 5 times faster than MATLAB's m-functions, which is why it wasn't shown in Fig. 13. When compared to C++, MEX/C++ files are generally slower due to requirement for conversion to special data types and transposed input and output arrays [11].

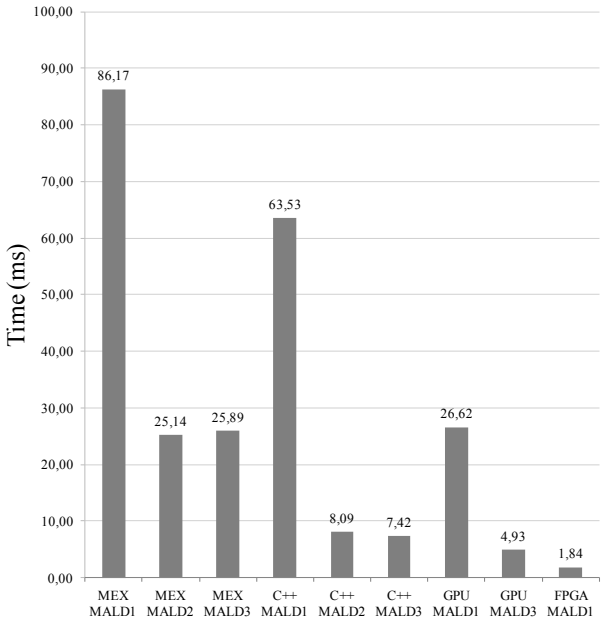


Fig. 12. Execution times of MALD methods implemented on different computational platforms, c.f. Tab. 1. Measurement is done on a ceramic tile image with the size of (1024, 1024) pixels and the kernel window size w = 32.

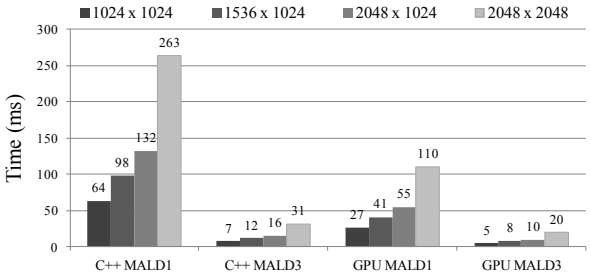


Fig. 13. CPU and GPU execution times of $MALD_1$ and $MALD_3$ with variable image sizes.

5 CONCLUSION

This paper proposes MALD method implementations on CPU, GPU and FPGA computational platforms. It is

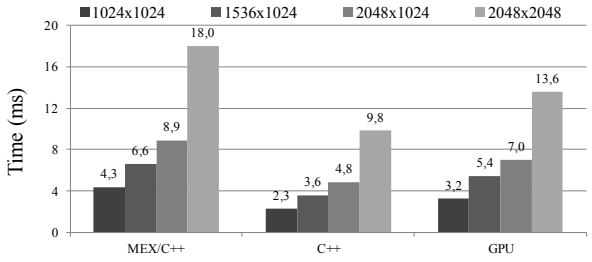


Fig. 14. $MALD_3$ integral image calculation time with varying image size.

shown that the best method, concerning ALU operation, is the proposed $MALD_3$ method, which uses integral image approach. Integral images enable mean value calculation in constant time and thus decrease the execution time of the MALD method mean value calculation. $MALD_3$ method has the least number of calculations and therefore has the slowest execution times on CPU and GPU platforms. GPU platform has better performance when compared to the CPU platform and the difference in execution time increases between the platforms in favor of the GPU as the image size becomes larger. FPGA platform is independent on the kernel window size and is only limited with the FPGA hardware resources. It gives the best result of all three platforms, but is the least flexible solution concerning further analysis. FPGA solution is adjusted for a 1D line-scan camera, while CPU and GPU are used with a 2D camera. In general, MALD method processes 1D image lines independently, what enables comparison of all three platforms. Finally, the results of this paper show how a specific algorithm works in a combination with a specific computational platform. By comparing proposed implementations, one can achieve significantly shorter execution times. With shorter image processing execution time, a ceramic tile production speed can improve.

REFERENCES

[1] V. Hocenski, Novi pristup smanjenju utjecaja keramičke industrije na okoliš temeljen na neuronskim mrežama, disertacija, Sveučilište u Zagrebu, Fakultet kemijskog inženjerstva i tehnologije, Zagreb, 2012.

[2] C. E. Costa and M. Petrou, "Automatic registration of ceramic tiles for purpose of fault detection," in Machine Vision and Applications, vol. 11, 2000, pp. 225-230.

[3] Z. Hocenski and T. Keser, "Failure detection and isolation in ceramic tile edges based on contour descriptor analysis," *Proc. Control & Automation, 2007. MED '07. Mediterranean Conference on*, placeCity-Athens, pp. 1-6, June 2007.

[4] Z. Hocenski, S. Rimac-Drlje, T. Keser, "Automatic Inspection of Defects in Plain and Texture Surfaces," *Proc. 5th IEEE Int. Conf. Intelligent Engineering Systems 2001, INES 2001*, pp.221-227, 2001.

[5] V. Hocenski, Z. Hocenski, S.Vasilic, "Application of Results of Ceramic Tiles Life Cycle Assessment due to Energy Savings and Environment Protection," *Proc. IEEE International Conference on Industrial Technology, IEEE ICIT 2006*, Mumbai, India, pp. 2972-2977, Dec. 2006.

[6] Ž. Hocenksi, T. Keser and A. Baumgartner, "A Simple and Efficient Method for Ceramic Tile Surface Defects Detection," *Proc. 2007 International Symposium on Industrial Electronics*, placeCityVigo, pp. 1606-1611, June 2007.

[7] T. Keser, Ž. Hocenski, V. Hocenski, "Intelligent Machine Vision System for Automated Quality Control in Ceramic Tile Industry", Strojarstvo, vol.51, 2010, pp.101-110

[8] T. Keser, Automatizirani inteligentni sustav za klasiranje keramičkih pločica, Sveučilište J.J. Strossmayera u Osijeku, Elektrotehnički fakultet Osijek, 2009.

[9] K. G. Derpanis, "Integral image-based representation, " in Department of Computer Science and Engineering York University Paper, vol. 1, 2007, pp. 1-6.

[10] F. Crow, "Summed-area Tables for Texture Mapping," *Proc. of SIGGRAPH '84*, placeCityMinneapolis, vol. 18, pp. 207-212, June 1984.

[11] I. Aleksi, D. Kraus, Z. Hocenski, "Multi-language programming environment for C++ implementation of SONAR signal processing by linking with MATLAB External Interface and FFTW," *ELMAR, 2011 Proceedings* , vol., no., pp.195-200, Sept. 2011.

[12] Z. Hocenski, I. Aleksi, R. Mijakovic, "Ceramic tiles failure detection based on FPGA image processing," *Industrial Electronics, 2009. ISIE 2009. IEEE International Symposium on* , vol., pp. 2169-2174, 5-8 July 2009.

[13] Xilinx user guide: "Power Methodology Guide," UG786 (v13.1) March 1, 2011.

[14] Naresh Grover, M. K. Soni: "Reduction of Power Consumption in FPGAs - An Overview,", I.J. Information Engineering and Electronic Business, pp. 50-69, 2012.

[15] W. W.-K. Shum, J. H. Anderson: "FPGA glitch power analysis and reduction," Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design (ISLPED '11), pp. 27-32, placecountry-regionUSA, 2011.

[16] A. R. Brodtkorba, T. R. Hagena, M. L. Sćtrab, "Graphics processing unit (GPU) programming strategies and trends in GPU computing, " in Journal of Parallel and Distributed Computing, vol. 73, Viktor Prasanna, Ed. Elsevier, 2013, pp. 4-13.

[17] T. Matić and Ž. Hocenski, "Parallel Processing with CUDA in Ceramic Tiles Classification, " *Proc. 14th international conference on Knowledge-based and intelligent information and engineering systems KES'10*, placeCityCardiff, pp. 300-310, 8-10 Semptember 2010.

[18] NVIDIA Corporation, "NVIDIA CUDA C Programming Guide Version 4.0," http://developer.download.nvidia.com/compute/cuda/4_0/toolkit/docs/CUDA_C_Programming_Guide.pdf, accessed 27 January 2012.

[19] B. Fulkerson, S. Soatto, "Really Quick Shift: Image Segmentation on a GPU," in Trends and Topics in Computer Vision, vol. 6554, Springer-Verlag, 2012, pp. 350-358.

[20] M. Gipp, G. Marcus, N. Harder, A. Suratanee, K. Rohr, R. König, R. Männer, "Haralick's texture features computed by GPUs for biological applications," in International Journal of Computer Science, vol. 36, International Association of Engineers, 2000, pp. 127-137.

[21] M. Backer, J. Tünnermann, B. Mertsching, "Parallel k-Means Image Segmentation Using Sort, Scan and Connected Components on a GPU," in Facing the Multicore-Challenge, vol. 7686, 2013, pp. 108-120.

[22] A. Abramov, T. Kulvicius, F. Wörgötter, B. Dellen, "Real-Time Image Segmentation on a GPU," in Facing the Multicore-Challenge, vol. 6310, 2011, pp. 131-142.

[23] Z. Yuan, W. Si, X. Liao, Z. Duan, Y. Ding, J. Zhao, "Parallel computing of 3D smoking simulation based on OpenCL heterogeneous platform," in The Journal of Supercomputing, vol. 61, 2012, pp 84-102.

[24] V. K. Nimmagadda, A. Akoglu, S. Hariri, T. Moukabary, "Cardiac simulation on multi-GPU platform," in The Journal of Supercomputing, vol. 59, 2012, pp. 1360-1378.

[25] V. Galiano, H. Migallon, V. Migallon, J. Penades, "GPU-based parallel algorithms for sparse nonlinear systems," in Journal of Parallel and Distributed Computing, vol. 72, 2012, pp. 1098-1105.

[26] S. Tomov, R. Nath, H. Ltaief, J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," *Proc. 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*, Atlanta, pp. 1-8, April 2010.

[27] G. E. Blelloch "Pre?x Sums and Their Applications," in Computer, vol. 9, 1990, pp. 35-60.

[28] M. Harris, S. Sengupta, J. D. Owens, "Parallel prefix sum (scan) with CUDA", GPU Gems 3, Addison Wesley, ch. 39, 2007.

[29] CUDPP Documentation, http://www.gpgpu.org/static/developer/cudpp/rel/cudpp_1.0a/html/index.html, accessed 26 January 2012.

[30] NVIDIA GeForce 9800 GT, http://www.nvidia.com/object/product_geforce_9800gt_us.html, accessed 26 January 2012.

[31] D. Kirk and W. Hwu: "Programming massively parallel processors," Morgan Kaufman Publishers, Elsevier 2010.

Tomislav Matić was born in 1983 in Brčko, Bosna and Herzegovina. In 2007 he graduated from the Faculty of Electrical Engineering, University of Osijek, Croatia where he got a lab technician position. He is currently a Ph.D. student at the same Faculty, where he works as a research assistant at the Department of Computing and Sofware Engineering. Teaching activities include undergraduate and graduate courses such as Digital electronics and Reliability of Computer Systems. His research interests include image processing, GPGPU computing and parallel computing. He is the co-author of two teaching books used for Digital electronics and Computer system design courses. He is a member of the IEEE.

Ivan Aleksi was born in 1982. He received PhD in technical sciences from the Faculty of Electrical Engineering, University of Osijek. At the same faculty his teaching activities include undergraduate and graduate courses such as computer architecture and sonar engineering. His research interests include sonar data processing and simulation for purpose of underwater machinery reconstruction. He is particularly interested in the implementation of parallel data processing on the graphics card processors. He is a co-author of two Faculty books used for the course of Computer Architecture. He is a member of KoREMA and IEEE.

Željko Hocenski (1952) is working as a scientist and researcher in the area of industrial electronics, computer engineering, automation and process control. He received B.Sc. (1976), M.Sc. (1984) degree in Electrical Engineering and Ph.D. degree (1996) in Computing from the Faculty of Electrical Engineering and Computing (FER Zagreb), University of Zagreb, Croatia. He was working at the Institute of Electrical Engineering of holding Končar in Zagreb (1977-1984) in the fields industrial electronics and automation, microprocessor control and industrial communications. From 1984 is at J.J. Strossmayer University of Osijek, Faculty of Electrical Engineering, position full professor (2006). He was Vice-dean (1997-2003) and Dean (2003-2005). Now is the head of the Computer Engineering Department. Teaching activities include undergraduate and graduate courses. Scientific activities are in design, diagnosis, verification and validation of embedded computer systems as well as fault-tolerant computer systems for process control and automation. His research results are published in more than 80 scientific articles and conference papers, books, course-books, 20 studies/reports and projects related to production. Prof. Hocenski has particularly excelled in the leadership of national and international research projects. Awarded as author of three technical improvements and innovations. He is a member of KoREMA (Managing Board), IEEE Computer, Signal Processing, Control, Education (also Croatian section), ACM and SICE.

AUTHORS' ADDRESSES

Tomislav Matić, assistant,
Ivan Aleksi, Ph.D.
Prof. Željko Hocenski, Ph.D.
Computer and Software Engineering Department,
Faculty of Electrical Engineering,
Josip Juraj Strossmayer University of Osijek,
Kneza Trpimira 2B, 31000 Osijek, Croatia,
email:tmatic1@etfos.hr, ivan.aleksi@etfos.hr,
zeljko.hocenski@etfos.hr

Received: 2012-07-04
Accepted: 2013-11-08