A Parallel Quantum Histogram Architecture

G. M. Megson, J. O. Cadenas, *Member, IEEE*, R. S. Sherratt, *Fellow, IEEE*, P. Huerta and W.C. Kao, *Senior Member, IEEE*

***Abstract*— A parallel formulation of an algorithm for the histogram computation of *n* data items using an on-the-fly data decomposition and a novel quantum-like representation (QR) is developed. The QR transformation separates multiple data read operations from multiple bin update operations thereby making it easier to bind data items into their corresponding histogram bins. Under this model the steps required to compute the histogram is *n*/*s* + *t* steps, where *s* is a speedup factor and *t* is associated with pipeline latency. Here, we show that an overall speedup factor, *s*, is available for up to an eightfold acceleration. Our evaluation also shows that each one of these cells requires less area/time complexity compared to similar proposals found in the literature.**

***Index Terms*— Histogram, parallel architectures, parallel algorithms, quantum representation.**

I. Introduction

THE advancement of image sensor technology tends to produce higher resolutions for digital cameras. Once a picture is taken it requires compression, possible watermarking, and transmission. Histogramming techniques are used in various forms in all these processes and thus a fast histogramming technique is of great interest to designers of consumer based electronics.

In this paper we propose a novel encoding that allows implicit parallelism to be exploited to produce a range of fast efficient architectures. Currently, camera designers can only use down-sampled image data to reduce the time of histogram analysis. If the histogram analysis can be performed while the image is being read from the sensor, the histogram can be evaluated in real time. This is an important feature, for example, in the auto exposure function since the scene brightness measurement should be done before the next exposure cycle starts. Also, some other enhancement techniques such as auto-level stretching, colour saturation enhancements (all typical stages in image processing pipelines of digital cameras [1]) become easier to perform while reducing time significantly. Current techniques such as tone reproduction are sometimes bypassed because the current system-on-chip (SoC) solutions cannot meet real-time requirements [2]. The algorithm and architectures presented here are suitable for full integration within a camera sensor thus allowing real-time analysis to become possible.

In a recent paper we proposed an array of cells to compute the histogram in a parallel and pipelined fashion [3]. These architectures can accept multiple pixels at a time and update multiple histogram bins at a time without the need of explicit memory blocks which makes the histogram computation fast. Further implementations in FPGA technology of these architectures found that the cells become too complex for processing more than 4 pixels in parallel per clock cycle [4]. This paper takes a different approach and follows a parallelisation of the generic serial histogram algorithm as the starting point. As a result of the consequent reformulation and manipulation of the algorithm, the internal structure of the cells is simplified compared to [3, 4] and consequently array architectures for processing 8 pixels per clock cycle become feasible. The twist behind the reformulation is that we recode the histogram algorithm using a quantum-like representation (QR); this produces a separating function that can be easily parallelised.

The quantum histogram algorithm is presented in Section II. Sketches of architectures for the computation of histogram using the QR representation are given in Section III while implementation in ASIC technology is given in Section IV with reference to a specific example. The paper ends with a brief discussion and some final conclusions in Section V.

II. Quantum Histogram Algorithm

To compute the histogram for a non-negative set of *n* integers where the data values correspond directly to the histogram bins (such as pixels in a grey-scale image) it follows that:

for *i*=1 **to** *n* **do**
  histogram[*data*[*i*]]= *histogram*[*data*[*i*]]+1

It is assumed item *data*[*i*] takes any value in the range 0, ..., *m*-1 (for a data item of *k*-bits then $m = 2^k$), and computation proceeds in $O(n)$ time steps. A step is the cost of an increment and two look-up operations. In essence, the histogram accumulates a count for each *data*[i] into one of *m* bins (contained as array *histogram*). The bin accumulated is the one labelled with a tag number equal to the *data*[i] value itself. Direct synthesis of the algorithm using space-time transformations or re-timing [5] is not straightforward because the index *data*[*i*] of the histogram creates a run-time dependency. This means that any data flow graphs cannot be

G. M. Megson is with the School of Electronics and Computer Science, University of Westminster, London W1T 3UW, UK (e-mail: g.megson@westminster.ac.uk).

J. O. Cadenas and R. S. Sherratt are with the School of Systems Engineering, The University of Reading, Reading RG6 6AX, UK (e-mail: {o.cadenas, r.s.sherratt}@reading.ac.uk).

P. Huerta is with Escuela Técnica Superior, Universidad Rey Juan Carlos, Madrid, Spain (e-mail: pablo.huerta@urjc.es).

W. C. Kao is with Department of Applied Electronics Technology, National Taiwan Normal University, Taipei, Taiwan (e-mail: jungkao@ntnu.edu.tw).

known until run-time. To overcome this problem we either place bounds on the size of *data*[*i*] and work with bounded graphs where the routing information and control signals are constrained or attempt to reformulate the algorithm into something easier to handle. In the former case this restricts the parallel structures implicit in the method and so the types of architecture that can be generated.

Our approach here is to reformulate the method in a novel way to expose and develop more parallelism. This reformulation step is often counter-intuitive because it essentially slows the sequential algorithm in order to produce an algorithmic form in which the data dependencies will accelerate the parallel implementation. The way this is achieved is by introducing redundancy and temporary storage to effectively break-up and manipulate the critical data dependencies. In our case this intermediate step will result in a faster overall outcome.

A. Quantum representation

First, consider a quantum formulation of a basic data item to be recognized by the histogramming method (such as a pixel). Suppose we have access to the bits encoding *data*[*i*], since *data*[*i*] < *m* the number of bits is $\log_2 m$ and so *data*[*i*] = $d_{k-1}d_{k-2} \ldots d_0$ where $k = \log_2 m$.

Define these bits of data as a *qubits* with the form:

$$d = a_0 \begin{bmatrix} 1 \\ 0 \end{bmatrix} + a_1 \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad (1)$$

where a_i are probability amplitudes which in this particular case need only take the values a_0=1, a_1=0 for bit=0 and a_0=0, a_1=1 for bit=1 [6]. The qubits for each data entry *data*[*i*]$_j$ = d_j defines the j^{th} bit as a 2-element vector. The tensor operator onto two *qubits* is then defined as:

$$d_i \otimes d_j = \begin{bmatrix} a_0^i \\ a_1^i \end{bmatrix} \otimes \begin{bmatrix} a_0^j \\ a_1^j \end{bmatrix} = \begin{bmatrix} a_0^i a_0^j & a_0^i a_1^j & a_1^i a_0^j & a_1^i a_1^j \end{bmatrix}^T \quad (2)$$

which is easily extended to any number of bits such that, in general

$$data[\mathrm{i}] = d_0 \otimes d_1 \otimes \ldots \otimes d_{k-1} = \left[0 \cdots 0\ e_{data[i]} 0 \cdots 0\right]^T \quad (3)$$

Note $e_{data[\mathrm{i}]} = 1$ so that the Quantum Representation (or QR), maps each unique *data*[i] to a unique *m*-dimensional vector such that for two distinct data items *data*[*i*] and *data*[*j*], the inner product of the vectors are orthogonal and for two identical values *data*[*i*], *data*[*j*] the inner product is unity. This forms a separating function where each of the *n* data items can be mapped to a space of dimension *m* such that each unique data point (or pixel) lies on one of the basis vectors spanning the space. The basis vectors correspond to bins in the histogram.

B. Quantum histogram

A serial histogram algorithm based on the QR formulation can be defined as follows:

for *i*=1 **to** *n* **do**
  r = *data*[*i*]$_0 \otimes$ *data*[*i*]$_1 \otimes$ *data*[*i*]$_{k-1}$; // see (3)
  for *j* = 0 **to** *m*-1 **do**
    histogram[*j*] = *histogram*[*j*] + *r*[*j*]

from which it is clear that the algorithm is decomposed into two steps; the process of reading the input *data*[*i*] and the process of choosing and incrementing the bins. Note that the calculation of vector *r* contains implicit parallelism while the *j*-loop performing the increment contains redundancy because only one *r*[j] item will be non-zero. Both these elements are hidden in the sequential formulation by the indirect index which forms the run-time dependency. This separation allows the splitting of the histogram assembly process into a qubit maker stage operating on the input data, a QR calculation to form vector *r*, and the increment of bins.

Another way to think about the construction is in terms of a data-dependency graph [7]. A graph for the sequential computation cannot be drawn until run-time when the data values are known. In the Quantum histogram formulation these unknown elements have been rolled back off the graph to form the inputs to the tensor required to construct *r*, so that a static data dependency graph can be constructed. Thus we can think of the tensor as a black-box that takes the data in and outputs, *r*. The architecture inside the box is parameterized by the number of bits, *k*, of a data item not the data elements so its data dependency graph is static for a given data range (*m*). Likewise since there is essentially no dependency between iterations of the *j*-loop for the increment step, the iterations can be computed in parallel. Notice that the operation to compute *r* is a binary associative operator. Hence, it can be formulated as a series of unrolled inner step style computations or as a fan-in operation based on tree architectures [8, 9]. Consequently it can be formulated either in linear mode or as a binary tree which we omit for brevity. Instead we use a binary decoder as the architecture which will be explained later.

III. Architecture for Quantum Histogramming

Since *n* >> *m* by definition an architecture for a given size *m* size can be built by pipelining the number of data items from which the histogram is constructed as discussed in section 2. An architecture to compute the histogram for *data*[*i*] of *k* bits is represented in block diagram as shown in Figure 1.

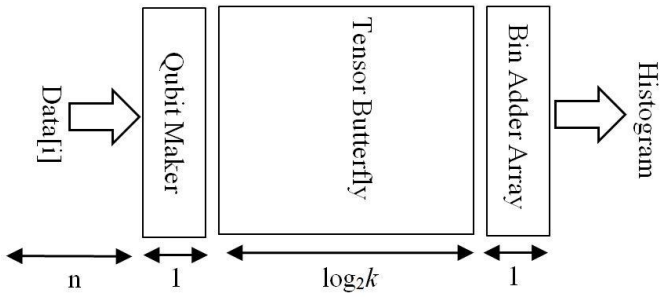

Fig. 1. Generic quantum histogram architecture for $m = 2^k$ bins.

The whole histogram computation for a small case of *m* = 4 bins; two bits for *data*[i] is shown in Fig. 2.

A. Quantum pipelined histogram cell

To create a design that is faster than the original sequential method requires further parallelism. We also note that the generation of the quantum representation suffers from scalability issues with the decoder (or butterfly arrangement) expanding exponentially as a function of the data bit-length. For example, m = 256 possible data items requires an 8 bit representation so that the previous architecture requires 8 qubits input vertically which are then expanded by the tensor butterfly into a 256 element vector with a single element set.

Controlling scalability requires a bound on the maximum number of bits to represent a piece of data. This can be achieved by a simple quantization on the input to provide a more abstract or second level of binning. In the case of m = 256 (consecutive) elements suppose we choose b = 4 second

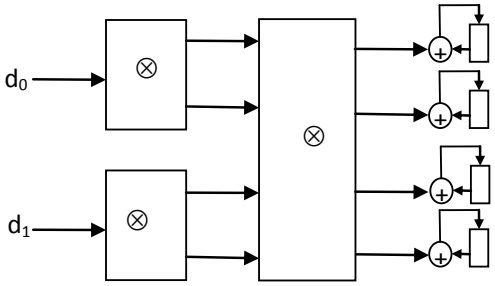

Fig. 2. Quantum histogram architecture for m = 4 bins, logic filled with ⊗ are AND gates.

level buckets. Each bucket captures 64 elements. And, in general bucket j, captures data items numbered (j-1)*(m/b) +1 to j*(m/b). It follows that the items in any bucket can be represented by the range 1 … (m/b) using a suitable offset. For convenience, it is easier to number the bucket range from zero so that the number of bits required is $\log_2 L = \log_2 (m/b) = \log_2 m - \log_2 b$. In the example this yields 6 = 8-2 bits to cover a bucket. Thus $L = m/b$ defines the number of bins per bucket. In this example with L of 6 bits, this implies computing (3) for j = 0, 1, …, 4 on each bucket for each data item.

A quantum histogram can be pipelined by a linear array of cells. A cell suitable for pipelined operation is shown in Figure 3. The array consists of b cells and each cell reduces the input moving left to right until the value fits into the range of a bucket. The QR tensor is then used to select which item is chosen within the data set of the bin. For a case of m = 256, and partition size L = 32 an array will be composed of $m/L = b$ = 8 cells. Each cell would guard 32 locations in r. In general an array of b cells computes the histogram on n data items in $T = n + (m/L) - 1$ time steps. Since the number of bits required to encode the data range L controls the size and latency of the butterfly this can be scaled by choosing the number of cells.

B. Parallel quantum pipelined histogram cell

A quantum histogram cell capable of managing multiple input data items is shown in Figure 4. This cell accepts two data items in parallel or s = 2. It is a form of double-pipeline array with the top half similar to Figure 3 and the bottom half a mirror image.

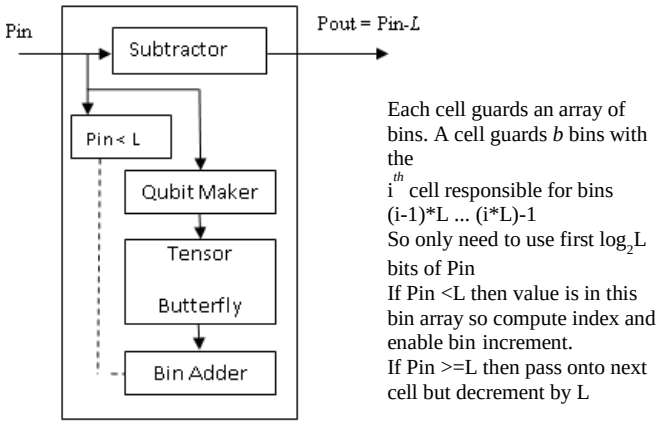

Fig. 3. A quantum histogram cell suitable for pipelined operation in an array.

This is essentially unrolling the algorithm back in Section II.B by a factor of two, or index i updated by two in the outer *for* loop. This recognizes that there are no data-dependencies between the n data items used to construct the histogram. The two parts of the cell are linked by the bin adder where it is possible that both tensor arrays will attempt to increment the same bin. Consequently, a bin is incremented by 0, 1, or 2 and we can use the bits emerging from the tensors to form control bits to choose the correct increment. This principle can be extended to higher degrees of pipelining but the increment arbitration becomes more complex. Hence we describe it for only the double pipe arrangement.

The double pipe means that the time to compute the histogram on a set of n data items is essentially reduced by half compared to a single pipelined computation. In general, if a cell can accept s data items in parallel, a speedup factor of s is achieved. Note that for the special case of L = 1 there is only one qubit and so the tensor butterfly can be removed making the cell simpler. For s = 2, an array computes the histogram in $T = n/2 + m/2 - 1$ and it is clear that we can scale up to L adders per bin by adding in the tensor butterfly to generate 2L control bits rather than two (a pair for each adder in the bin). For the case of $L = m$ the array reduces to a single cell with one tensor butterfly controlling an adder bin of n adders. This, in fact, is one of the first schemes to compute the histogram found in the literature [11] and still popular in hardware [12]. So we see that the quantum formulation effectively parameterizes a range of histogram architectures trading off area and time.

IV. Quantum Histogram ASIC Implementation

An implementation for the parallel computation of the histogram follows straightforwardly by extending the cells presented in Figs. 3 and 4. For this discussion, a cell will be processing s data items (labeled *P*in) per clock cycle with each data item represented as k-bits. The cell will be responsible for computing a constant of L bins with L being a power of two. Within the cell each data item needs to be compared against L. To compute all $m = 2^k$ bins we need m/L cells with each cell occupying a position b from 1 to m/L within the array.

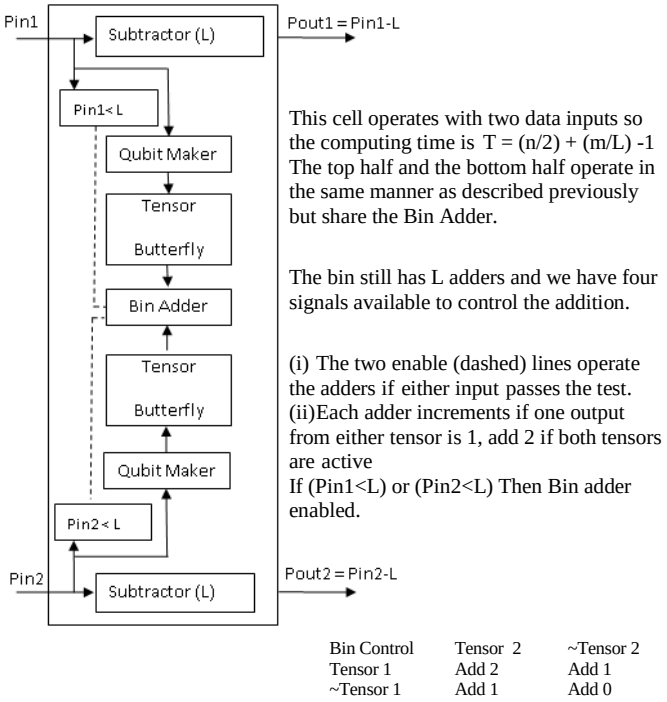

Fig. 4. A quantum histogram cell capable of accepting two data items in parallel.

As each cell passes *P*out = *P*in –*L* to the neighboring cell, a cell with position *b* receives an input which has already been discounted by the amount (*b*-1)**L* (*P*out should be allowed to become negative). As a consequence the comparison of *P*in to *L* only needs to check the *k*-log$_2$*L* MSB of *P*in and see if they are all zero.

The log$_2$*L* LSB bits of *P*in are used to generate a QR internal to the cell corresponding to this smaller block of bits. When any comparison e_q (for q = 1, …, s) becomes true it indicates that a bin within the cell must be updated and consequently that comparison bit is used as an enable to the decoder as seen in Fig. 5. When all comparisons e_q are false all decoders are disabled and no bin update can occur.

The QR bits (decoder's outputs in Fig. 5) are fed into the "Bin Adders" block where logic decides which (if any) of the bins must be updated and more importantly by which amount.

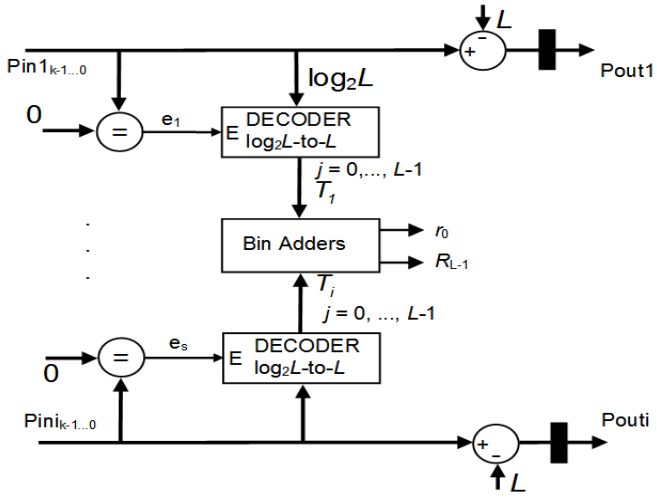

Fig. 5. Cell architecture for keeping L bins per cell.

For each bin accumulator, two things need to be decided; the value by which the bin must be updated and whether a given bin should be updated or not. The value to be updated is computed as $v_j = \sum_{i=1}^{s} T_{ij}$ while an enable for each register adder h_j is the Boolean OR of all T_i at position j. Suppose two data items are being processed in a cell that keeps bins values 4, 5, 6, and 7. If the same bin 5, is selected by the tensor then the output of each are [0 1 0 0] and [0 1 0 0] respectively, given a value v_1 = 2 only for bin 5 (notice h_1 = 1).

TABLE I
AREA AND FREQUENCY FOR HISTOGRAM CELLS ARCHITECTURE

s	L=2		L=4		L=8	
	Gates	MHz	Gates	MHz	Gates	MHz
s = 2	583	236	980	222	1749	220
s = 4	779	192	1210	176	2224	175
s = 8	1202	165	1650	150	2648	148
	Cells presented in [4]					
s = 2	738	228	1015	218	-	-
s = 4	1089	173	1386	159	-	-

Area (in gate count) and frequency (in MHz) results from ASIC TSMC 0.35µm (up to s = 8 data items per clock cycle and up to L = 8 bins per cell).

If the values were 5, 6 the tensors output [0 1 0 0] and [0 0 1 0] respectively giving v_1 = 1 and v_2 = 1 with h_1 = 1 and h_2 = 1 as expected. However, the tensor bits might have been disabled at the decoder, there is no need to keep the h_j signals in the architecture provided the logic makes v_j = 0 for the case when no bin should be updated.

Table 1 shows area and frequency results in ASIC technology of a quantum histogram cell of Figure 5 when processing 2, 4 and 8 data items per clock cycle (*s*); for each *s* value the cell can compute either of 2, 4 and 8 bins per cell (*L*). The table also includes the area/time to the cell designs presented in [4] for comparison. Thus, parallelism can be exploited for speedup gain (increasing *s*) or for reducing latency (increasing *L*) or both at the expense of extra circuitry. Note from the results that either speedup can be doubled or latency can be halved without doubling the cell's area. Calculating the histogram computing time in *msec*/*Mpixel*, gives around 2.1, 1.3 and 0.8 for *s* = 2, 4, 8 respectively; this implies a sustained time reduction as *s* increases without being affected by the number *L* of bins per cell, suggesting the cell's structure scales well in terms of hardware.

V. DISCUSSION AND CONCLUSION

The internal cell architecture capable of processing *s* data items per clock cycle as shown in Fig. 5 is composed of simple and well understood logic blocks interconnected in a structured manner. This allows a more regular design than our previous processing cell presented in [4]. This is especially appreciated when capturing RTL code for the cell, an important step before synthesis of circuits. This regularity is maintained even when the number of bins per cell increases. There are *s* binary comparators each of log$_2$*L* bits to generate enable signals *e* in Fig. 5. Also, there are *s* binary decoders each of *L* outputs and *L* accumulators, one for each bin in the cell.

Synthesis of full arrays for the histogram computation of 256 bins is reported in [3, 4] for up to four data items per clock cycle and up to four bins per cell. As shown in Table 1,

with the presented cell architecture, synthesis of cells capable of processing up to eight data items per clock cycle while keeping up to eight bins per cell were obtained in a very straightforward manner. In particular the qubit generation and tensor butterfly has been replaced by fast binary decoders using $\log_2 L$ bits [10]. This optimization is illustrated in Fig. 5. Furthermore, the bin control bits can be implemented efficiently as carry-save-adder (CSA) tree structures [13]. The generation of h_j signals do not incur extra logic since they become available as part of the addition process embedded in either half or full 1-bit adders required for the CSA tree or can be removed altogether.

Within a cell, each bin accumulator operates in a truly parallel manner as made explicit by the on-fly quantum representation derived from a block of incoming data bits. The critical path is essentially dominated by the adder for the accumulator plus the CSA tree. This follows because the comparison and QR generation is relatively shallow in the level of logic gates owing to the fact that it is dependent on the number of bits in the encoding not the number of data items.

As expected, the more parallelism in a cell the longer the critical path becomes. Due to the regularity of the cell, two-level pipelining (see [14]) can be easily put into place within the cell to meet more stringent clock frequency (this is to be reported elsewhere). A histogram array composed of cells as presented here can operate in streaming mode as data is directly collected from a source such as a high resolution camera sensor. For a speedup gain of s, we only require a sensor to supply s data items per clock cycle. This avoids accessing internal memory buffers multiple times before histogram can commence which is typical in solutions that perform the histogram as a post-processing step once the data has been stored.

Once a histogram is completed on a block of data the array can be primed to restart histogramming on a new block of data, also in a streaming manner, using the same mechanisms presented in [4]. The cell in [3] requires handcrafting Boolean logic functions for each s, L parameter; for $s = 2$ $L = 2$ these are equations of three Boolean variables, but for $s = 8$, $L = 8$ this would require equations of up to 25 Boolean variables. The logic for the cells here is better structured.

Recently, a parallel array histogram architecture was presented in [15]. No pipelining is used and instead a full $k{:}2^k$ binary decoder is required to avoid latency of an array. The design here essentially becomes the design in [15] when $L = 2^k$ or when all the histogram bins are bound to a single bucket, thus this work is a more general formulation by offering design parameters to play with performance/latency trade-offs to meet requirements of a particular application.

In conclusion, with the reduced complexity of the quantum-based histogram cells, it becomes very practical to synthesize circuits capable of accepting up to eight data items per clock cycle thus allowing for an eightfold speedup factor for the computation of histograms. This speedup is obtained without doubling the cell area compared to previous designs. It is also feasible to maintain up to eight bins per cell favouring shallower histogram arrays and consequently lower latencies. The ability to process eight bits means that the arrays deal in data sizes that are meaningful (e.g. pixels) in the larger context of the computation. Given the regularity of both cells and arrays the histogram computation can be integrated closer to the camera sensor thus enabling a real-time analysis.

REFERENCES

[1] W. C. Kao, S. H. Wang, L. Y. Chen, Y. Lin, "Design considerations of color image processing pipeline for digital cameras," *IEEE Trans, Consumer Electron*. 52 (2006) 1144-1152

[2] C. T. Chiu, et. al., "Real-time tone-mapping processor with integrated photographic and gradient compression using 0.13um technology on an ARM SoC platform," *J. on Signal Proc. Syst*. 64 (2011) 93-107.

[3] J. Cadenas, R.S. Sherratt, P. Huerta, "Parallel pipelined histogram architectures," *Elecron. Lett*. 47 (2011) 1118-1120.

[4] J. O. Cadenas, R. S. Sherratt, P. Huerta, W. C. Kao, "Parallel pipelined array architectures for real-time histogram computation in consumer devices," *IEEE Trans. Consumer Electron*. 57 (2011) 1460-1464.

[5] C. Leiserson, F. Rose, J. Saxe, "Optimizing synchronous circuits by retiming," *3rd Caltech Conf. on VLSI*, 1 (1993).

[6] M. A. Neilsen and I. L. Chuang, *Quantum Computation and Quantum Information*, Cambridge University Press, Cambridge, 2000.

[7] U. Banerjee, "An introduction to a formal theory of dependence analysis, Springer," *The J. of Supercomputing*, 2 (1988), 133-149.

[8] S. Muchnick, *Advanced Compiler Design Implementation*, Academic Press, London, 1997.

[9] T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to Algorithms*, 2nd Ed. The MIT Press, Cambridge, 2003.

[10] J. F. Wakerly, *Digital Design Principles and Practice*, 4th ed., Pearson Prentice Hall, New Jersey, 2006

[11] S. Muller, "A new programmable VLSI architecture for histogram and statistics computation in different windows," *Proc. Int. Conf. on Image Processing*, DC, USA, (1995), pp. 73-76.

[12] E. Garcia, "Implementing a histogram for image processing applications," Xcell Journal Online 38, Xilinx (2000), pp. 46-47.

[13] B. Parhami, *Computer Arithmetic, Algorithms and Hardware Designs*, Oxford University Press, New York, 2000.

[14] H.T Kung, L. M. Ruane, D. W. L. Yen, "Two-level pipelined systolic array for multidimensional convolution," *Image and Vision Compu*. 1 (1983) 30-36.

[15] Q. Gan, J. M. P. Langlois, Y. Savaria, "Parallel array histogram architecture for embedded implementation," *Elecron. Lett*. 49 (2013) 99-101.