Real-Time Hough Transform on 1-D SIMD Processors: Implementation and Architecture Exploration

Yifan He[1,2], Zoran Zivkovic[1], Richard Kleihorst[1],
Alexander Danilin[1], Henk Corporaal[2], and Bart Mesman[2]

[1] NXP Semiconductors, the Netherlands
{yifan.he,zoran.zivkovic}@nxp.com
[2] Technische Universiteit Eindhoven, the Netherlands
y.he.1@student.tue.nl, h.corporaal@tue.nl

Abstract. In the first part of this paper, an improved slope-intercept like representation is proposed for implementation of Standard Hough Transform (SHT) on SIMD (Single-Instruction, Multiple-Data) architectures with no local indirect addressing support. The real-time implementation is realized with high accuracy on our Wireless Smart Camera (WiCa) platform. The processing time of this approach is independent of the number of edge points or the number of detected lines. In the second part, we focus on analyzing the differences between the SHT implementations on 1-D SIMD architectures with and without local indirect addressing. Three aspects are compared: total operation number, memory access/energy consumption, and memory area cost. When local indirect addressing is supported, the results show a considerable amount of reduction in total operations and energy consumption at the cost of extra chip area. The results also show that the focuses for further optimization of these two architectures are different.

1 Introduction

The Hough Transform (HT) [1, 2] is a well-established and robust algorithm to locate straight lines in images in presence of noise and occlusion. In general, two sets of parametric representation are used for Standard Hough Transform (SHT): slope-intercept based [1], and (ρ, θ) based [3]. The main difference of the first representation with respect to the second is its unbounded parameter space.

The Hough Transform can cope with noise, gaps in outlines and partial occlusion, even in complicated backgrounds. However, the implementation of SHT requires massive computation, large memory space and high bandwidth. Without parallel processing on a proper platform, it can be hardly implemented in real-time, especially with high accuracy on high-resolution images. Fortunately, the SIMD architecture fits all the requirements. The intrinsic parallelism due to the pure pixel-based feature of the HT makes it extremely suitable for this kind of architecture.

In the first part of this paper, an improved slope-intercept like representation for SHT is proposed, which is a very efficient way to implement SHT on 1-D SIMD architectures with no Local Indirect Addressing (LIA) mode (i.e. all processor elements access the same memory row at one time). The real-time implementation is realized with high accuracy on our Wireless Smart Camera (WiCa) platform [4], which is a powerful image/video processing platform, containing Xetal [5] (a 1-D SIMD processor) as the

J. Blanc-Talon et al. (Eds.): ACIVS 2008, LNCS 5259, pp. 254–265, 2008.
© Springer-Verlag Berlin Heidelberg 2008

main component (see Fig. 1 and 2). The processing time of this approach is independent of the number of edge points or the number of detected lines. With the proposed novel Hough Space (HS) structures and the efficient image "rotation" mechanism, all lines can be detected in real time with VGA format (30 fps) as input.

After that, the proper SIMD architectures for SHT are exploited. We focus on comparing the differences between SHT implementations on 1-D SIMD architectures with and without LIA. Total operation number, memory access/energy consumption, and area cost are elaborated in detail. The results show that when the LIA is supported, a considerable amount of reduction in total operations and energy consumption is achieved at the cost of extra chip area (see Section 3.4). We also show that the further optimization focuses for these two architectures are different. The architecture without LIA benefits a lot from parallel processing of the processing array and control processor, while the architecture with LIA requires extra logic (see Section 3.5) to reduce the overhead for preparing the edge points.

2 The Efficient SHT Implementation

The complete HT for line detection can be divided into a set of subtasks: a) edge detection; b) voting; c) Hough Space post-processing; d) detected line displaying. The Canny operator [6] is used in our implementation for edge detection. As this step is not the focus of this paper, only steps b) ~ d) are discussed.

2.1 Basic Implementation

Proposed Equation: $b = x + y*m$ for 1-D SIMD Processors without LIA

We propose the $b = x + y*m$, $-1.0 \le m \le +1.0$ representation according to the characteristics of the 1-D SIMD architecture. Note that, here m-b does not have the same meaning as the standard slope-intercept representation mentioned in Section 1. Comparing with Li's method [7], one obvious improvement in this representation is that only a one-cycle MAC (Multiple-Accumulate) instruction is required instead of multiple cycles due to the removal of the tgθ calculation. The calculation accuracy is also improved (Note numbers are represented by limited number of bits in hardware. More calculation on the same data means more accuracy loss. This is especially visible in an embedded processor.)

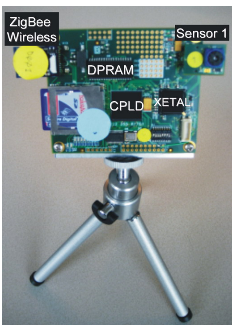

Fig. 1. The WiCa Platform

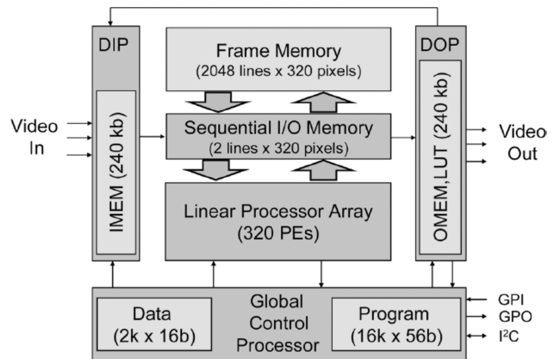

Fig. 2. Block Diagram of the Xetal-II Architecture

In our method, a line-based fashion is adopted instead of a multi-pass processing on the same image like Li did [7]. HS is stored in the line memory of the SIMD processor, and each line memory in the HS stands for a “Hough Line”, which has the same m value but different b values in each cell. The voting procedure is done as follows: When an input video line is received (after edge detection step, the pixel value is either 0 or 1), we start from m = 0, which merely adds the video line to the corresponding Hough Line (since b = x). This process costs only 1 cycle in the Xetal processor. After that m is increased by Δm (say 0.02, roughly 1°), and new shifting amount (h-y)*m is calculated. Here h is the image height, and y is the index of current row. We call it a “hitting the bottom” method, as all the edge points on the same line will vote to the cell, whose x coordinate is the same as that of the cross point where the line hits the image bottom (See Fig. 3). (Note that we can also use y*m. Then the physical meaning becomes “hitting the top” instead of “hitting the bottom”). The same video line is shifted to the right (for m > 1), and a copy is shifted to the left (for m < 0). Votes (the shifted video line) are added to the corresponding Hough Line. This process is continued till m reaches the last value (1.0). Then a new video line is read in, and the current video line is sent to the LCD screen for displaying, or stored for other following processing.

After processing the last video line, the whole HS is built in the line memory, and post-processing can be applied on it. As we only process a limited number of m values with step Δm, a practical line could fall in between two processed m values with its voting spreading among several neighboring cells instead of concentrating into one. These lines can still be captured by adding the votes of cells in HS to their neighboring ones. With local peak detection and a global threshold, target lines are detected. The final step is to display the detected line on the LCD screen. An inversed process of the voting procedure is used. Instead of shifting the input video lines, Hough Lines are shifted to reproduce the detected lines. With this method, lines can be displayed on the screen efficiently, independent of the number of total detected lines.

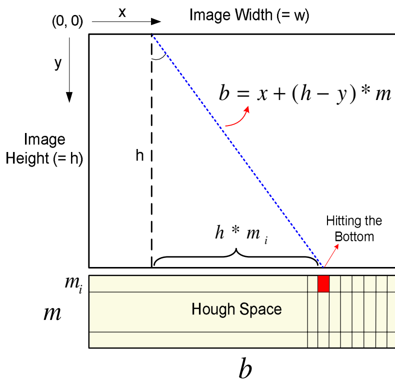

Fig. 3. Proposed implementation method on an SIMD processor

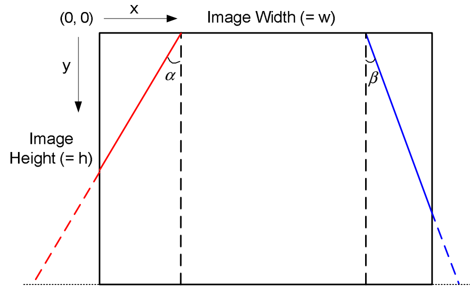

Fig. 4. Some lines are “missing” if we only apply the basic approach

2.2 Capture All Lines in Every Half Plane

By applying the basic implementation on a frame and its 90°-rotated version (Twin HT [8]), all lines are expected to be detected. The precondition is that every pass must

be able to capture all of the lines in its half-plane (i.e. [-45°, +45°] for the original pass, and the other half for the rotated pass). However, this precondition does not hold true if we only apply the basic method introduced in Section 2.1. Fig. 4 gives an example: Two lines, which are within the range of [-45°, +45°], are “missing” as they can not “hit” the image bottom.

In order to solve this problem, a property of lines in the half-plane is exploited: for any line between -45° and +45° in an image of size N×N, it will either “hit” the top or the bottom border, or “hit” both. Thus, a “hitting the top” concept is also introduced together with the “hitting the bottom” concept. For every pass, Dual Hough Spaces are built, which are called “Top HS” and “Bottom HS” respectively. Fig. 5 depicts this idea. Using this approach, the problem described in Fig. 4 can be solved. Another solution we proposed is *Interleaved Hough Space*, refer to [9] for details.

2.3 Twin Hough Transform with Efficient Image “Rotation”

In order to implement the full-plane HT, only one extra pass on the 90°-rotated image is required. However, rotation of a whole image usually costs too much time, which makes it a main bottleneck in the twin HT concept. In our implementation, a very efficient “rotation” approach is proposed. The WiCa platform, as well as many other embedded systems, contains an off-chip memory to supply extra data storage. An efficient image rotation can be achieved with the help of this off-chip memory. During processing the first pass on the input frame, the video lines are stored into this memory with a normal write mode (store the frame from left to right, and top to bottom). This is done in parallel by the control processor, so the transfer time is hidden to the HT. When processing the other pass on the 90°-rotated image, image data is read back with a different mode (from bottom to top, and left to right). Thus, the rotated image data is acquired without really rotating the whole image in the memory. This “rotation” mechanism is very efficient, as the transfer time for reading is also done in parallel with the remaining part of the program. In [9], more information about the full implementation and applications are discussed.

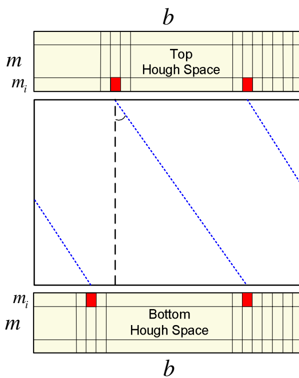

Fig. 5. Using both “hitting the top” and “hitting the bottom” methods

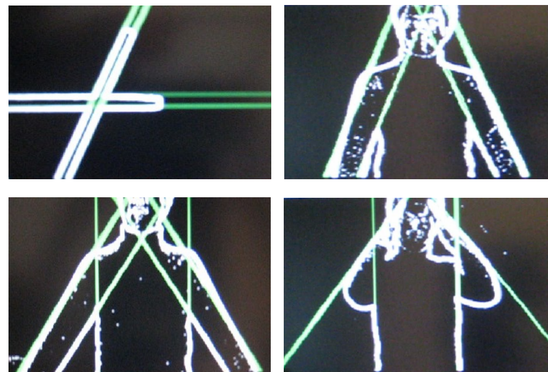

Fig. 6. These are the snapshots from the real-time demos. Detected lines are overlaid with green color on the (white) edge image.

Thus, the full implementation of SHT is realized on the WiCa platform. Fig. 6 presents some snapshots from the real-time demos on our WiCa platform. The proposed implementation has the following features: 1) One instruction to generate the vote; 2) LIA is NOT required (to support local indirect addressing, both area and complexity of the SIMD processor increases due to the extra logic); 3) Content Independent (processing time are independent of the number of edge points or detected lines); 4) Efficient Shifting and Image "Rotation"; 5) Efficient Line Displaying, which is independent of the number of total detected lines.

3 1-D SIMD Architecture: Global or Local Indirect Addressing

As a follow up research of [9], we focus on exploring the proper 1-D SIMD architecture for SHT in this paper. For an SIMD processor, it is a fundamental design choice whether to support LIA or not. When the LIA is supported, different PEs can access different memory rows at one time, which means more data-access freedom. However, the hardware requirement is that every PE needs its own memory bank and address generator (AG), which increases the chip area (see Section 3.4). If LIA is not supported (Global Indirect Addressing, GIA), then only one AG and a single big memory bank are required. All PEs can only access the same memory row at one time, which is sufficient enough for many low-level image processing algorithms (e.g. filter), but could be less flexible for others. Fig. 7 depicts the two architectures.

Hough Transform can be implemented on both architectures based on their different characteristics. In this section, we will analyze the differences between the two SHT implementations on them. Three aspects are compared: total operation number, memory access/energy consumption, and memory area. Optimization focuses are also proposed based on the comparison results.

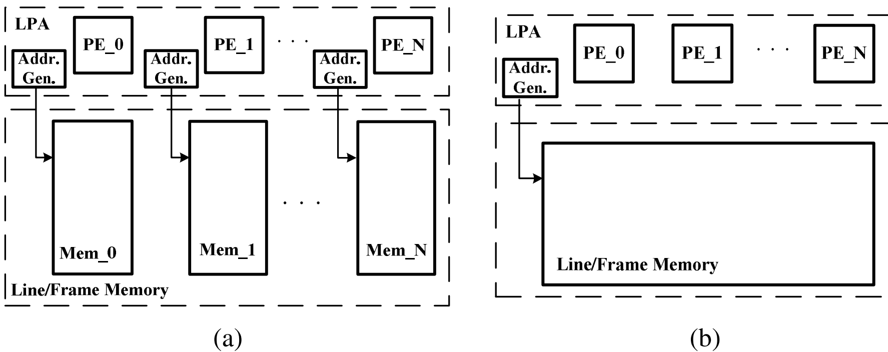

Fig. 7. (a) Local Indirect Addressing; (b) Global Indirect Addressing

3.1 SHT on 1-D SIMD with LIA

In this section, we will briefly introduce the implementation method when LIA is supported. As only the edge/feature points contribute to the HT, we can broadcast the

coordinates of the edge points one by one to each PE, then apply the HT (either $<b, m>$ or $<\rho, \theta>$ space), schematically shown in Fig. 8. Suppose $<\rho, \theta>$ space is used, the procedure is shown as follow: At the initial step, each PE is assigned a different θ value, say θ_j. The calculated ($cos\theta_j$, $sin\theta_j$) pair is sent and stored in the local registers of each PE. After that, the HT step starts. The coordinates of the edge point i (x_i, y_i) are broadcasted to each PE. ρ_i is calculated according to the formula $\rho_i = x_i *cos\theta_j + y_i*sin\theta_j$. Then the value ρ_i is mapped to the row address, where the corresponding Hough Cell is kept. The final step is to update the Hough Cell by adding 1.

A similar method ($<\rho, \theta>$ space) is used in [10] to map SHT onto IMAP. Here, we show that with LIA, no matter what kind of parameter space is chosen ($<b, m>$, $<\rho, \theta>$, or any other), the SHT implementation are identical. They can be considered as the derivation from the general formula given below, in which f and g are two functions of one parameter, and h is the other parameter. The only difference in the implementation code is the initial step: For $<\rho, \theta>$ space, different ($cos\theta_j$, $sin\theta_j$) pairs are stored, while for $<b, m>$ space, 1 or different m_i are stored.

General Formula: $h = x * f + y * g$

$<\rho, \theta>$ *space:* $h = \rho_i, f = \cos\theta_i, g = \sin\theta_i$

$<b, m>$ *space:* $h = b_i, f = 1$ (half of the PEs), m_i (others),
$g = m_i$ (half of the PEs), 1 (others)

Before comparing and analyzing the two different implementations and architectures, some assumptions are made, which are listed below:

- Pixel values of input image is either 0 or 1 (after edge detection)
- Same size of HS (same overall granularity)
- Comparison is based on *Basic Operations*
- *Hardware Loop* is used in the control processor
- *Delay Slot* caused by instructions like jump, load, etc. can always be filled
- Read and write access to the memory dissipates the same amount of energy

Edge detection is considered as a preprocessing step. Here the bit width of the pixel is irrelevant. The second assumption is also valid. The comparison should be based on the same overall accuracy. Though we have Xetal architecture and instruction set as our reference, the comparison on basic operations (ALU, Load/Store, Branch, etc.)

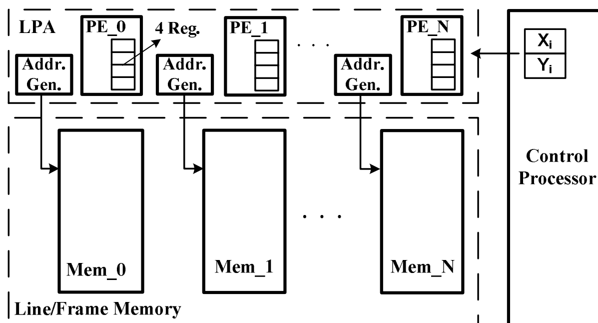

Fig. 8. SHT Implementation (LIA Supported)

is still preferred. Otherwise, different ideas about possible combination and optimization would complicate the analysis, which leaves the results less convincing. Hardware-Loop is widely used in the modern DSP processors [11]. Xetal-II also supports hardware loop. With this technique, a constant-iteration loop has zero overhead. The fifth assumption does not always hold, as there are not always enough independent instructions to be filled in the delay slot. However, we still consider it valid for simplicity. In practice, memory read and write consume different amounts of energy. But for high-level estimation, they are usually not differentiated, especially when they appear in pairs (as in our case).

The symbols used in the following sections are listed here.

- N: Number of processing elements (PE)
- M: Number of different m values
- P: Percentage of the edge points in an image

Comparison and analysis below start with the image resolution of N×N (N is the same as the number of PEs). Both algorithms scale equally for higher resolutions. Suppose the high resolution is kN × kN (k is an integer), and the same detection accuracy is required. For the first implementation (GIA), the image can be firstly divided into k^2 N×N same parts, then processed one by one. So both the operations and memory accesses become k^2 times. For the second implementation (LIA), the total operation count and memory accesses also increase k^2 times, as they depend on the total number of pixels.

3.2 Number of Operations Comparison

Xetal-II is used as the reference architecture in the following comparison. The total number of PEs is 320 (N = 320), data is 16-bit wide, and frame memory has 2048 lines. Fig. 9 presents the main code of SHT implementation (basic Hough Space) when only GIA is supported. Operations with prefix 'G_' are Global Control Processor (GCP, scalar processor) operations; the others are Linear Processor Array (LPA, vector processor) operations. The outermost loop repeats for every image line. The second loop is for different values of parameter m. The innermost (Label_1) is the shifting loop. For a specific |m| value, the shifting distance (dist) is calculated. Then two copies of the input image line either shift to the right (-m) or to the left (+m) until the destination is reached (shift = dist). On average, the total shifting distance per input image line equals to N/2.

The full implementation requires two passes of Dual-HS together with image "rotation". The image "rotation" is negligible (see Section 2.3). Thus, the total operation number for a whole image is given by equation (1). When M = 160 (the same HS granularity as the other implementation) and N = 320, the total operation count is 2,201,600.

$$Op_{GIA} = (15M + 14N) * N \tag{1}$$

When LIA is supported, the approach introduced in Section 3.1 is chosen, which broadcasts the coordinates of the edge points. Fig.10 depicts the processing for the edge points, which is the general code (the practical code could be slightly different).

So, for every edge point, there are only 9 operations. However, this does no include the selection of edge points and maintaining the coordinates. We will show later that these non-HT parts even dominate the total number of the operations. Fig. 11 presents

```
// LOOP from 1 to ROW_END (ROW_END = N)                                  // Operations/image line
// hardware-loop is supported, zero delay                                   Basic-HS    Dual-HS
{
  // initial part per image line (negligible)
  LdLine   image_lineL,  input_line   // copy current line                  1           1
  LdLine   image_lineR,  input_line   // copy current line                  1           1
  G_Set    m ,           INIT         // initialize m                       1           1
  G_Set    shift,         0           // set variable shift to 0            1           2

  // LOOP from 1 to M/2 (M is the total number of different m values)
  // hardware-loop is supported, zero delay
  {
    G_Mul    dist,   m,    row_index  // shift distance                     M/2         M
                                      // row_index is current row number
    Label_1:
    {
      ShiftL   image_lineL            // shift image line 1 pixel to the left    4*N/2   4*N
                                      // two operations here: load and store
      ShiftR   image_lineR            // shift image line 1 pixel to the right
                                      // two operations here: load and store
      G_Add    shift,  shift,  1      // increase shift amount              3*N/2       3*N
      G_Cmp    shift,  dist           // compary if variable shift = dist
      G_JmpLT  Label_1                // if shift amount is less than dist
    }
    // update Hough Space
    LdLine   ACCU, hl_a               // load correspoding hough line       6*M/2       6*M
    Add      ACCU, ACCU, image_lineL  // add
    StLine   hl_a, ACCU               // store correspoding hough line
    LdLine   ACCU, hl_b               // load correspoding hough line
    Add      ACCU, ACCU, image_lineR  // add
    StLine   hl_b, ACCU               // store correspoding hough line
    G_Add    m,      m,    STEP_M     // update m                           M/2         M/2
  }
  G_Add    row_index, row_index, 1    // increase row_index                 1           1
}
```

Fig. 9. Main code of SHT when only GIA is supported

```
// HT body                                            // Operations/edge-point
{
  // send coordinates                                     2
  Send      xi,   x
  Send      yi,   y
  // h = x*f + y*g                                        2
  Mul       ACCU, xi,   f        // x * f
  Mac       res,  yi,   g        // res = y*g + ACCU
  // Address linear mapping: addr = h*a + b               2
  Mul       res,  res, CONST_a   // res = h * a
  Add       addr, res, CONST_b   // addr= res + b
  // update Hough Space                                   3
  Load      res, [addr]
  Add       res,  res,  1
  Store     res, [addr]
}
```

Fig. 10. Main code of SHT when LIA is supported (edge-point part)

the whole piece of the code. According to it, the overhead is 7 operations. The total number of operations for a whole image is given by equation (2). When N = 320, Op_{LIA} is 716,800 + 921,600*P.

$$Op_{LIA} = 16*P*N^2 + 7*(1-P)*N^2 \quad (2)$$

Fig. 12 depicts the comparison of the operation number between these two implementations (GIA and LIA). It shows that when LIA is supported, the total amount of operations is reduced significantly, even when all pixels are edge points (P = 100%). This is mainly because when only GIA is supported, an extra loop is required to cover all different m values. And the innermost shifting loop costs many operations too (Note that LIA is independent of the number of m or ρ values). When P = 5% (a typical case),

```
// LOOP from 1 to N*N (last pixel)
// hardware-loop is supported, zero delay                    // Operations
{
  G_Load    pixel, [P_addr]       // load pixel                       3
  G_Cmp     pixel, 1              // compare if or NOT an edge point
  G_JumpNE  Lable_1               // Not an edge point, omit HT

  // HT body, 9 operations/edge-point
  {   ...   }

  Label_1:
    G_Add     x,     x,     1     // update x                         3
    G_Cmp     x,     ROW_END+1    // last pixel in the line
    G_JumpNE  Label_2

    // these 2 ins. are only called once per image line, negligible
    G_Set     x,     1            // set x to 1
    G_Set     y,     y,     1     // set y to the next line
  Label_2:
    G_Add     P_addr, P_addr, 1   // update pixel address             1
}
```

Fig. 11. Main code of SHT when LIA is supported (whole)

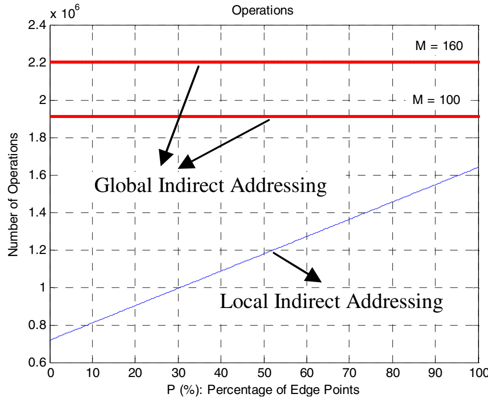

Fig. 12. Number of operations for both implementations

Op_{GIA} is 2.9 times of Op_{LIA}. Another important result is that in the implementation with LIA, more than 80% of the operations are consumed by the non-HT part.

3.3 Memory Access and Energy Consumption Comparison

Both methods need to read the input image, this is a negligible part compared to the total memory energy consumption. So we focus on the memory energy consumption cost by the HT part. According to Fig. 9 and 10, the number of memory accesses of the first implementation (GIA, N*16bit read or write) is defined by equation (3), while the number of memory access of the second implementation (LIA, 16bit read or write) is defined by equation (4).

$$MA_{GIA}\ (N*16bit\ read/write) = 8*N^2 + 8*M*N \tag{3}$$

$$MA_{LIA}\ (16bit\ read/write) = 2*N*P*N^2 \tag{4}$$

The depth of memory bank is set to 2048, the same as we have in Xetal-II. The memory energy consumption for LIA (single-port SRAM, under CMOS90LP technique and

84MHz) is 40.1pJ/access (16bit read/write). As memory with word-width of 5120 (320*16bit) has to be custom-designed, and the energy consumption depends on the process, here we choose the practical solution used in Xetal-II as a comparison. In Xetal-II, every 8 PEs are assigned a memory bank. So a total of 40 memory banks with word-width of 128bits are used. The energy consumption of a 128bit×2048 memory bank is 179.2pJ/Access under the same condition. So when access the same amount of data, 44% of energy is saved. If we also take the energy consumption on AGs and many address wires of LIA into consideration, the number is even bigger (Less energy consumption per unit access is an important merit of GIA). We can save more energy/unit-access when wider word-width is used. However, the speed for further reduction slows down rapidly according to the data we have. This is also one of the reasons that we use 128bits memory bank in Xetal-II.

Though GIA consumes less energy/unit-access, the implementation of SHT on it still consumes more total energy. This is because the total amount of memory accesses of the implementation on GIA depends on the number of different m values, while the implementation on LIA does not. The total energy consumption of both implementations is given in the following equations. The comparison is depicted in Fig. 13 (N = 320). When P = 5% and M = 160, the implementation with GIA consumes 66 times more energy.

$$E_{GIA} \text{ (word-width = 128 bits)} = 179.2*N/8*(8*N^2 + 8*M*N) \text{ pJ} \quad (5)$$

$$E_{LIA} \text{ (word-width = 16 bits)} = 80.2*P*N^3 \text{ pJ} \quad (6)$$

3.4 Memory Area Comparison

Under CMOS90LP technique, 84MHz frequency, and N = 320, the total frame memory areas of these two architectures are: 25.92 mm^2 (LIA, word-width = 16 bits), and 18.68 mm^2 (GIA, word-width = 128 bits). The architecture with only GIA costs less memory area (28% less in this case). If the area of AGs and wiring is also taken into consideration, GIA can save even much more.

3.5 Optimization Focuses

From the analysis above, we found that when LIA is supported, the implementation requires fewer operations and consumes less energy. However, the area increases noticeably, as its memory layout (1 memory-bank/PE) costs more area, and the frame memory occupies a large percentage of the total chip area. Another interesting result we found is that the optimization focuses for the two implementations are different.

LPA/GCP in Parallel: Architectures with only GIA can benefit from parallel processing of Linear Processor Array (LPA) and Global Control Processor (GCP). This is because both LPA and GCP operations are required during processing HT. By analyzing the code in Fig. 9, a 34% reduction of processing time is seen when GCP and LPA are in parallel (when M =160). However, for SHT, architectures with LIA can hardly benefit from parallel processing of LPA and GCP, especially when P is small. This is because only edge points require LPA operations, and they usually account for a very small part. When P = 5%, only a 4.7% reduction is reached (refer to Fig. 10 and 11).

Optimization for Non-HT Part: We notice that when P is small, more than 80% of the operations for mapping SHT onto the SIMD processors with LIA are due to non-HT processing (picking up the edge points, and maintaining the x and y coordinates, see Fig. 11).

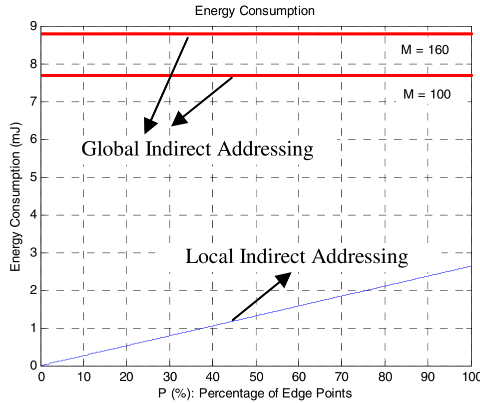

Fig. 13. Energy consumption for both implementations

So optimization should focus on reducing these non-HT operations. Techniques like automatic address increment can be used. Fig. 14 shows the ASIC logic for automatically increasing the load address, and for maintaining the pixel coordinates. With this extra logic, more than 60% of the processing time can be saved (P = 5%). Table 1 presents the overall comparison results. The final design choice depends on the target applications.

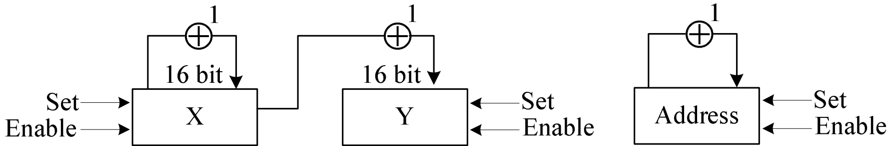

Fig. 14. Logics for maintaining the pixel coordinates and address

Table 1. Comparisons between global indirect addressing and local indirect addressing for SHT

	Pros	Cons
Global Indirect Addressing ($b = x + m \cdot y$)	1) independent of No. of edge points (constant processing time) 2) less area cost 3) less memory energy consumption per unit-access 4) fewer line memories for HS (same granularity) 5) operations considerably reduced when LPA/GCP in parallel processing	1) less data access freedom 2) dependent on No. of different m values 3) more operations required 4) more total memory energy consumption
Local Indirect Addressing ($h = x \cdot f + y \cdot g$)	1) more data access freedom 2) fewer operations required 3) less total memory energy consumption 4) operations dramatically reduced when non-HT part is optimized	1) dependent on the No. of edge points (non-constant processing time) 2) more memory area cost 3) more memory energy consumption per unit-access

4 Conclusions

In the first part of this paper, an improved slope-intercept like representation is proposed for implementing SHT onto SIMD architectures with no LIA support. The real-time implementation is realized with high accuracy on the WiCa platform, which is a powerful image/video processing platform developed by NXP Semiconductors. The processing time of this approach is independent of the number of edge points or the number of detected lines.

In the second part, we focused on comparing the design choice, GIA or LIA, from three aspects: total operation number, memory energy consumption, and memory area cost. GIA and LIA require different frame memory architectures. When the total memory size is the same, GIA occupies less area, and consumes less energy per unit-access. However, LIA provides more freedom for data accessing. For the comparison of mapping SHT, when LIA is supported, the results show a considerable amount of reduction in total operations and energy consumption, at the cost of extra chip area which is mainly due to larger frame memory. Moreover, the results also show that the focuses for further optimization for these two architectures are different. Architectures with only GIA can benefit a lot from parallel processing of LPA and GCP. But it does not help too much for LIA. Mapping SHT onto the SIMD processor with LIA can improve considerably if extra logic is used to reduce the non-HT operations. In this paper, we also showed that with LIA, no matter what kind of parameter space is chosen ($<b, m>$, $<\rho, \theta>$, or any other), the SHT implementation is identical.

References

1. Hough, P.: Method and Means for Recognizing Complex Patterns. U.S. Patent No. 3, 069, 654 (1962)
2. Illingworth, J., Kittler, J.: A Survey of the Hough Transform. Computer Vision, Graphics, and Image Processing 44, 87–116 (1988)
3. Duda, R., Hart, P.: Use of the Hough Transformation to Detect Lines and Curves in Pictures. Communications of the ACM 15, 11–15 (1972)
4. Kleihorst, R., Schueler, B., et al.: Architecture and Applications of Wireless Smart Cameras (Networks). In: IEEE Inter. Conf. on Acoustics, Speech and Signal Processing (2007)
5. Abbo, A., Kleihorst, R., et al.: Xetal-II: A 107 GOPS, 600mW Massively-Parallel Processor for Video Scene Analysis. In: International Solid-State Circuits Conference, pp. 192–201 (2008)
6. Canny, J.: A Computational Approach to Edge Detection. IEEE Transactions on Pattern Analysis and Machine Intelligence 8, 679–698 (1986)
7. Li, Z., Tong, F., et al.: Parallel Algorithms for Line Detection on a 1xN Array Processor. In: International Conference on Robotics and Automation (1991)
8. Risse, T.: Hough Transform for Line Recognition: Complexity of Evidence Accumulation and Cluster Detection. Computer Vision, Graphics, and Image Processing 46, 327–345 (1989)
9. He, Y., Zivkovic, Z., et al.: Real-Time Implementations of Hough Transform on SIMD Architecture. In: International Conference on Distributed Smart Cameras (2008)
10. Yamashita, N., Fujita, Y., et al.: An Integrated Memory Array Processor with a Synchronous-DRAM Interface for Real-Time Vision Applications. Pattern Recognition 4, 575–580 (1996)
11. Uh, G., Wang, Y., et al.: Effective Exploitation of a Zero Overhead Loop Buffer. In: ACM SIGPLAN workshop on Languages, compilers, and tools for embedded systems (1999)