Data Buffering and Allocation in Mapping Generalized Template Matching on Reconfigurable Systems

Xuejun Liang, Jack Jean and Karen Tomko
Department of Computer Science and Engineering, Wright State University, Dayton, OH 45435, USA

Abstract. Image processing algorithms for 2D digital filtering, morphologic operations, motion estimation, and template matching involve massively parallel computations that can benefit from using reconfigurable systems with massive field programmable gate array (FPGA) hardware resources. In addition, each algorithm can be considered a special case of a "generalized template matching" (GTM) operation. Application performance on reconfigurable computer systems is often limited by the bandwidth to host or off chip memory. This paper describes the GTM operation and characterizes the data allocation and buffering strategies for the GTM operation on reconfigurable computers. Several mechanisms that support different levels of parallelism are proposed and summarized in the paper. Finally, the implementation of an infrared automatic target recognition application on two commercial FPGA boards is used to demonstrate the various design options with different data allocation and buffering mechanisms and the pruning of the design space based on the FPGA area and memory constraints.

Keywords: Template Matching, Configurable Computing, Field Programmable Gate Array (FPGA), Reconfiguration

1. Introduction

Computing systems that use co-processor boards based on field programmable gate array (FPGA) chips may adapt their hardware resources to the application requirements. The technology has been demonstrated for the acceleration of various applications, such as automatic target recognition (ATR)[4, 9, 11], neural networks[1], Adobe Photoshop[7], Solar Polarimetry[8], and machine vision[6].

Image processing algorithms for 2D digital filtering, morphologic operations, motion estimation, and template matching share some common properties. They all involve massively parallel computations that can benefit from using massive FPGA hardware resources. In addition, each algorithm can be considered a special case of a *generalized template matching* (GTM) operation. As a result, designing optimal implementations for these operations can be characterized similarly and be solved systematically. A GTM operation consists of two steps as illustrated in Figure 1.

In the first step of the GTM, the input data are "convolved" (or "correlated") with a set of masks, or templates, to produce a set of convolved data. Note that an individual convolution may involve a sequence of masks and the computation associated with each mask may be *generalized convolution/correlation* as used in morphologic operations and in motion estimation. In the general case, the compu-

© 2000 *Kluwer Academic Publishers. Printed in the Netherlands.*

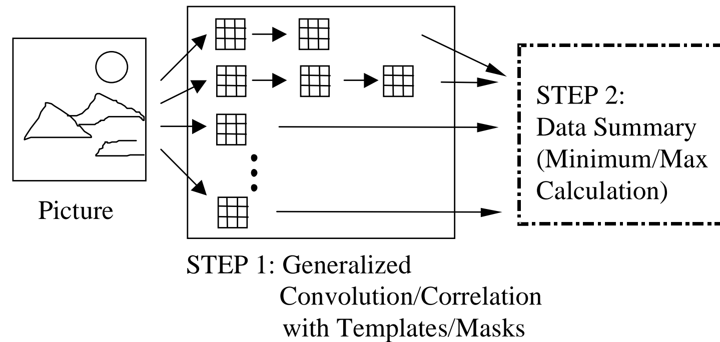

Figure 1. Generalized template matching

tation associated with each mask can simply be a pre-specified function on a number of pixels indicated by the mask. In the second step of the GTM, the convolved data are "summarized" which sometimes involves the calculation of the smallest value among a set of numbers. Each of the above mentioned algorithms is a GTM in the following sense.

1. *Template matching*: there is a set of masks in the first step. The second step summarizes each convolved data and chooses the best template based on some criteria. For example, the maximal value inside a convolved data area may be used to indicate the fitness of a template and the best template may be defined as the one that fits best, or equivalently, the one that has the largest maximal value. Template matching is frequently used in automatic target recognition applications[4, 9].

2. *Two-D digital filtering*: convolution of the masks and the image is used in the first step and there is no second step. If a sequence of cascaded filters is used, then there is a sequence of masks. If a filter bank is used, then there are multiple masks in the first step.

3. *Morphologic operations*: there is a sequence of masks in the first step and there is no second step. Generalized convolution is used for the computation where the multiplication and addition operations of a convolution are replaced with addition and maximal/minimal, respectively.

4. *Motion estimation*: there is a set of masks in the first step and a generalized correlation is used for the computation. The second step involves the calculation of the displacement vector for each mask. In this case, the set of masks may be parts of a previous image frame and they may have related values. In addition, the contents of masks may not be known until run time[2].

A special case of the GTM operation is the "Sliding Window-Based Operations" as in [10] where all the pixels (or samples) in the "window" (or mask) are involved

in the computation and sent to the computational units for processing. Note that in template matching sometimes a template is quite "sparse" and only a low percentage of pixels are involved in the computation. Such kind of template matching is a GTM operation that does not belong to that class of "Sliding Window-Based Operations."

This paper characterizes the mapping of the GTM operation on reconfigurable computers in terms of data allocation and buffering. Several mechanisms that support different levels of parallelism are proposed and summarized in the paper. Section 2 describes the mapping of a single mask to an FPGA chip with an external SRAM. The mapping of multiple masks to multiple FPGA chips is then described in Section 3. Section 4 presents a case study of utilizing data allocation and buffering strategies. Conclusions and future work is given in Section 5.

2. Mapping Single Mask to Single FPGA Chip

The following notation is used in this paper. For the GTM, suppose a mask is applied to an image. The image has r rows and c columns and each image pixel has b bits of precision. The mask has p rows and q columns. In addition the mask has w number of *active points* which include all the points necessary for the mask computation. (Usually a mask contains a pattern and the active points of a mask form the pattern.)

In this section, the FPGA board is assumed to have only one FPGA chip which is directly connected to one or multiple external memory modules. The image data may be sent from the host processor to the FPGA chip either directly or through the memory modules. The memory (or host) bandwidth to the FPGA can be a severe bottleneck for some applications.

When no image row is buffered inside the FPGA chip, each pixel needs to be read from outside the FPGA (the external memory modules or the host processor) w times. Since w is usually much larger than one, it is desirable to buffer enough number of image rows inside the FPGA chip so that each image pixel needs to be read only once. When the buffering of image data requires too much FPGA space, it becomes necessary to store image off chip and to access each image pixel multiple times. The following discussion includes three cases: (1) full buffering of image rows internally, (2) no internal buffering of image rows, and (3) a partial buffering scheme.

2.1. Full Image Row Buffering

Figure 2 shows a 3×4 mask being applied to an image where the mask covers twelve pixels, 0, 1, 2, 3, c, $c+1$, $c+2$, $c+3$, $2c$, $2c+1$, $2c+2$, and $2c+3$. If the pixels enclosed by the bold lines are buffered inside the FPGA chip, moving the mask one pixel right requires only the reading of one extra pixel, i.e., pixel $2c+4$, from outside the FPGA. In other words, when the FPGA chip can buffer $c(p-1)+q$ pixels, the image pixels fed into the chip can be fully reused internally and each external pixel

•	•	•	•	•	•	•	•	•	•
•	•	0	1	2	3	4	5	6	•
•	•	c	c+1	c+2	c+3	c+4	c+5	c+6	•
•	•	2c	2c+1	2c+2	2c+3	•	•	•	•
•	•	•	•	•	•	•	•	•	•

Figure 2. Pixels that need to be buffered for a 3×4 mask

needs to be read exactly once. That would greatly reduce the memory bandwidth requirement.

2.1.1. *Function-Level Parallelism*

The other advantage of internal buffering the image rows is that, when all the pixels required to evaluate one mask are available on chip, the parallelism at the function evaluation level can be explored. The adder tree used in [9] is an example.

2.1.2. *FPGA Buffers*

On Xilinx FPGA chips, there are three different mechanisms that can be used to buffer image rows.

1. Use the flip-flops in CLBs (Configurable Logic Blocks). One CLB can store two bits and those two bits can be accessed in parallel. This mechanism is good for storing the $p \cdot q$ pixels in a mask area so to support the function-level parallelism[9].

2. Use the RAMs that are based on function generators in CLBs. One CLB can store 32 bits (which cannot be accessed in parallel). This mechanism can be used to store the $(p-1) \cdot (c-q)$ pixels that are not currently used for the template.

3. Use the BlockRAM, which is available only on Xilinx Virtex chips. The XCV1000 chip has 32 blocks, each being a dual-ported 4,096 bit RAM. Therefore that chip supports the parallel accessing of 64 ports where each port can be up to 16 bit wide. The BlockRAM can be used in two ways to buffer pixels.

   – If CLB flip-flops are available to buffer the $p \cdot q$ pixels in a mask area, the BlockRAM can be used as a buffer so that in each clock cycle $p-1$ pixels are read from the BlockRAM and one external pixel fetched is written to the BlockRAM. One example with a 3×4 mask is shown in Figure 3(a) where consecutive pixels at the same column (such as 0, c, and $2c$) are assigned to form a vector of *stride* one. Here the stride value of a vector is the difference of block numbers between two consecutive vector elements. That guarantees the parallel reading of the $p-1$ pixels. In the diagram pixels 0, 1, 2, 3 in the BlockRAM are labeled with parentheses indicating that those pixels can be

(and could have been) overwritten. Note that consecutive pixels at the same row (such as c, $c+1$, $c+2$, $c+3$, ...) could have been assigned to the same block.

– If no CLB flip-flop is used for buffering pixels, the $c(p-1)+q$ pixels should be arranged so that the $p \cdot q$ pixels in a mask area can always be accessed from different ports. This can be done by (1) distributing consecutive pixels at the same row to form a vector of stride one, and (2) placing consecutive data at the same column to form a vector of stride q. One example with a 3×4 mask is shown in Figure 3(b).

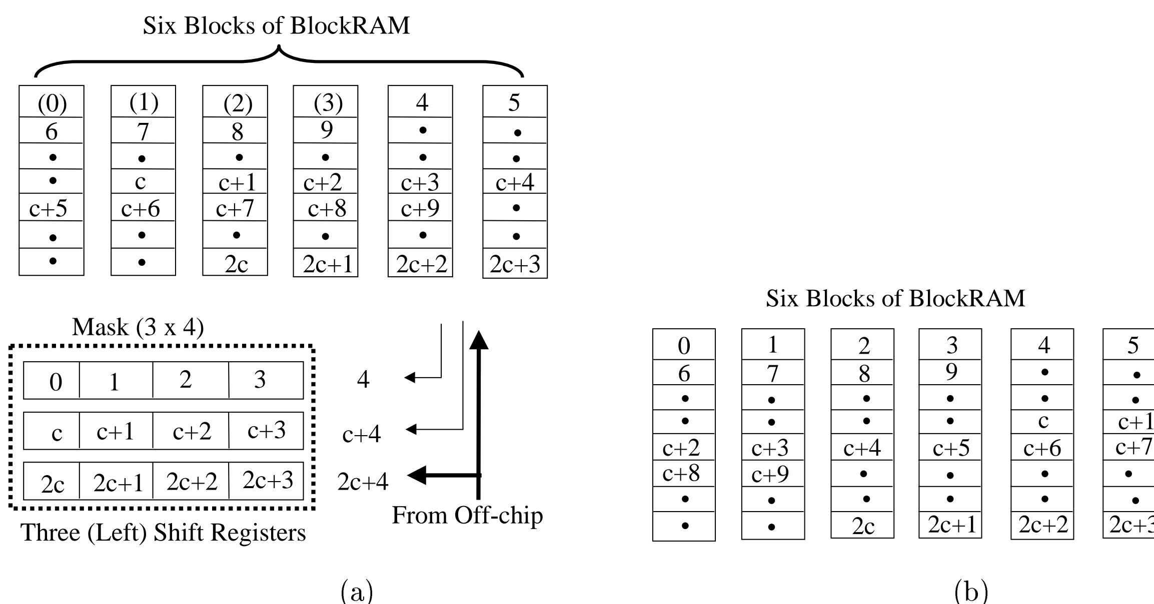

Figure 3. Using BlockRAM to buffer pixels: (a) With CLB flip-flops for shift registers, and (b) Without CLB flip-flops

Storing $c(p-1)+q$ pixels on chip can be very expensive. For example, if $p = 20$, $q = 20$, $c = 640$, and $b = 8$, there are 12,180 bytes to store. That translates into at least 3,045 CLBs on a Xilinx 4000 series FPGA chip. This solution is desirable when $(c(p-1)+q) \cdot b$ is small or when the properties of the computation allows for a bit sliced implementation (see [9] for an example).

2.2. No Buffering of Image Rows

When storing p image rows on chip becomes too expensive, the image can be stored off chip. The result is that each pixel needs to be read into the FPGA chip at least w times where w is the number of *active* points of a mask. If w is close to $p \cdot q$, then

it might be easier to simply read each pixel $p \cdot q$ times. However, if w is relatively small compared to $p \cdot q$, then it might be worthwhile to store the locations of the w active points inside the FPGA chip and access each pixel only w times. In this case, the external memory is accessed in a pseudo random sequence according to the active points of the mask. It takes w memory accesses to compute the application of the mask to one pixel location.

2.2.1. *Pixel-Level Parallelism*

Suppose each port of the external memory can provide k pixels at a time, those pixels can be consumed by using k copies of computation units so to compute the results of applying the mask to k consecutive pixel locations. Without loss of generality, the following description assumes $k = 2$ in the exploration of this pixel-level parallelism.

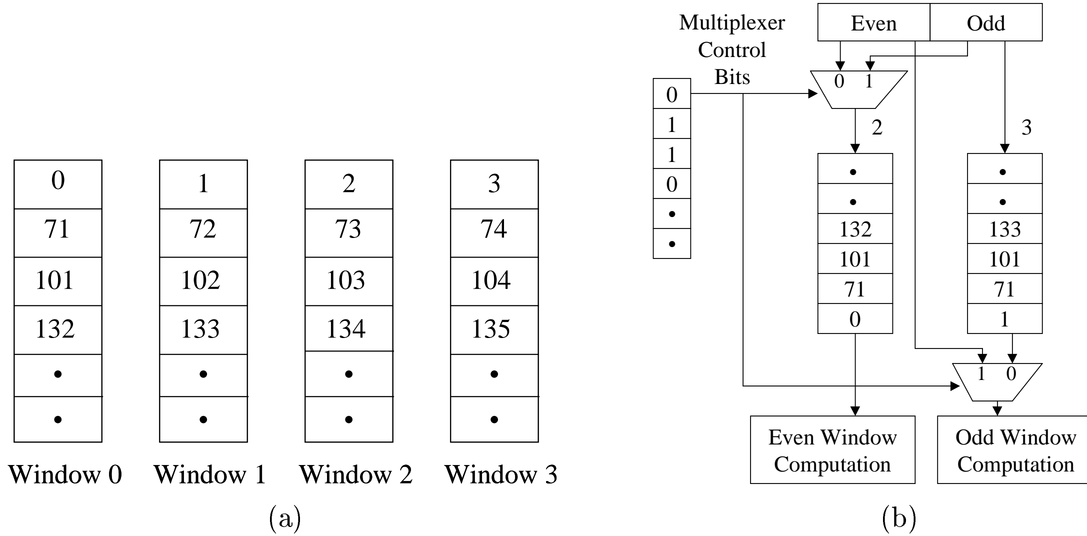

Figure 4. (a) Four windows resulting from applying a mask to four consecutive pixels, and (b) Using small internal buffers ($k = 2$)

Assume that pixels are stored in the order of scanned lines (row major) in the external memory and labeled in increasing integer numbers. Figure 4(a) shows four *windows* resulting from applying a mask to four consecutive pixel locations where the pixel numbers involved in applying a mask are indicated in each window. For example, for Window 0, the numbers 0, 71, 101, 103, and so on, indicate the active points of the mask, or equivalently the pixels that need to be accessed from external memory. When the mask is moved to the next pixel location, which results in Window 1, pixels 1, 72, 102, 133, and so on need to be used. To facilitate the parallel evaluation of windows 0 and 1, there is a need to provide simultaneously a pair of consecutive pixels n and $n + 1$ to those two copies of computation units where the values of n are 0, 71, 101, 132, and so on. When n is 71, providing 71

and 72 simultaneously requires special mechanism because the memory word size is equal to two pixels when $k = 2$, and a memory word always starts at an even pixel. The special mechanism can be either of the following.

1. *Redundant External Data Storage*: When pixels are stored in row major in the external memory, two neighboring pixels on the same image row cannot always be accessed in one clock cycle. For example, if pixels n and $n+1$ are stored at the same address, then pixels $n+1$ and $n+2$ are at two consecutive addresses and they cannot be accessed in one clock cycle. To facilitate the parallelism so that a mask can be evaluated on two neighboring pixels in parallel, the external memory stores redundant data as shown in Figure 5(a). In this scheme one of the two pixels stored at each address is redundant. That is, if pixels n and $n+1$ are stored at one address, then pixels $n+1$ and $n+2$ are stored at the next address. In this way, the external data storage matchs very well to the needs of the two copies of computation units. In [4, 3] such a design was used for an automatic target recognition application and implemented on a Giga Operations's G900 FPGA board. The overhead of storing redundant data is in the extra memory space required and the time to arrange and store the data.

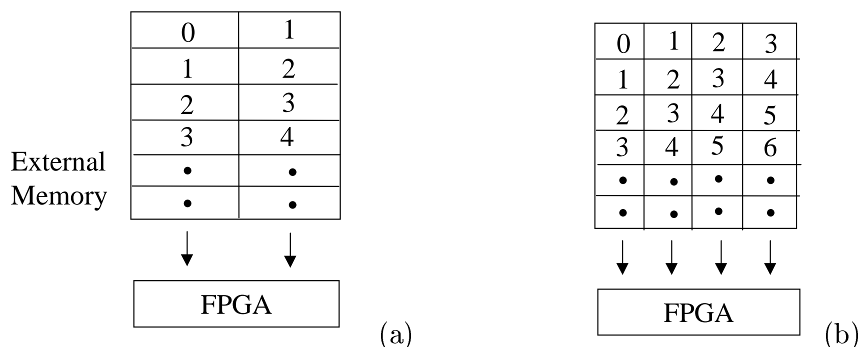

Figure 5. Using redundant external data storage: (a) $k = 2$, and (b) $k = 4$

2. *Small Internal Buffer*: The other option is to store pixels in the external memory in the order of scanned lines and to use a small internal buffer in the FPGA chip that can store w image pixels. One such buffer is required for each of the k copies of computation units. One example of such a design is shown in Figure 4(b). In this design the FPGA chip each time reads two image pixels from the external memory, one with an even address and the other with an odd address. There are two copies of the computation units, one dedicated to even windows and the other to odd ones. In the first clock cycle, pixels 0 and 1 are fetched as desired and stored without being consumed. (They will be consumed w cycles later.) In the second cycle, even though pixels 70 and 71 are fetched, only pixel 71 is useful and stored. Pixel 72 will not be available until w cycles later. The internal buffer therefore introduces w cycles of delay.

For many FPGA boards, a memory port is either 32-bit or 64-bit wide. Therefore it is very possible that k is either 4 or 8. In that case, the extension of the redundant storage scheme is straightforward while the extension of the internal buffering scheme leads to extra complexity in internal control logic. An example when $k = 4$ is shown in Figure 6. It is also possible to use a hybrid scheme that mixes the two schemes. An example when $k = 4$ is shown in Figure 7.

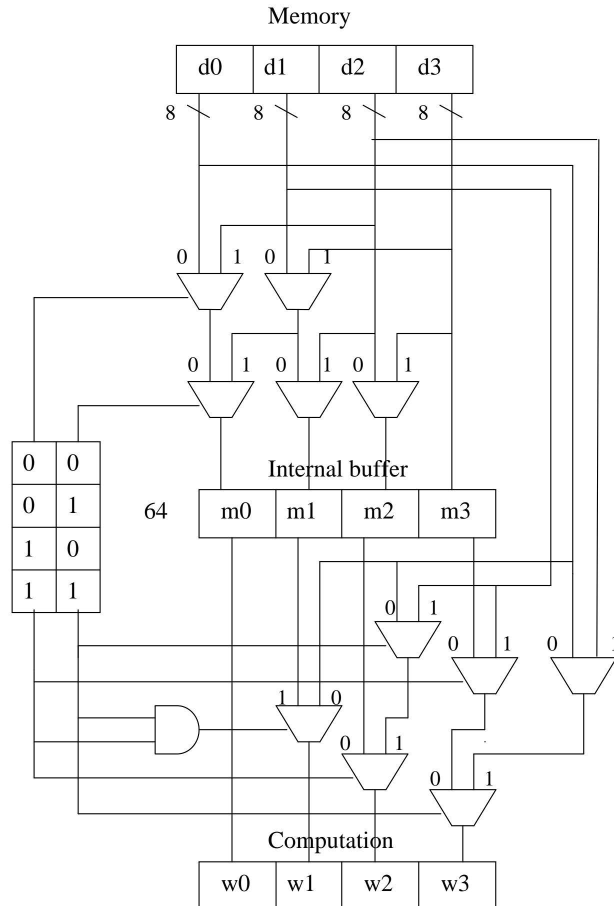

Figure 6. Internal buffer ($k = 4$)

2.2.2. *Multiple Memory Ports*

For some FPGA boards, such as the StarFire board from Annapolis Micro Systems, Inc., each FPGA chip is connected to multiple memory ports. For example, consider an FPGA chip that is connected to four different ports, two 32-bit wide and two 64-bit wide. In this case, it is possible to fetch four independent pixels in one clock cycle and therefore function-level parallelism can be explored even without the internal buffering of image rows.

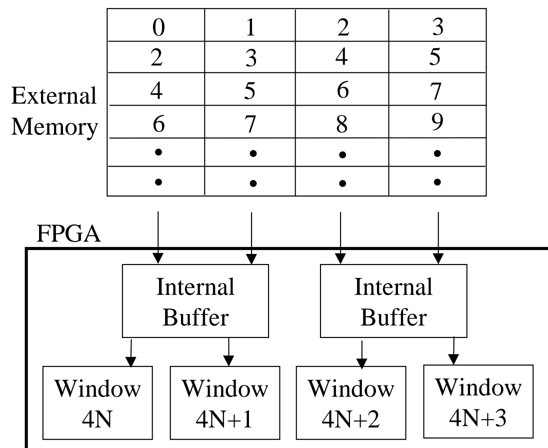

Figure 7. Using a hybrid scheme ($k = 4$) with redundant external data storage and small internal buffers

2.3. Partial Buffering of Image Rows

When storing p image rows on chip becomes too expensive, the image can be stored off chip. That does not rule out the possibility of buffering less number of image rows inside an FPGA chip. With full buffering, only one pixel need to be brought in from off chip per window evaluation. With no row-buffering, w pixels need to be accessed instead. By using partial buffering, only those active points that are not available on chip needs to be fetched. It is therefore possible to trade-off space and time and to optimize designs in this way.

3. Mapping Multiple Masks to Multiple FPGA Chips

When there are multiple masks, they may be evaluated in parallel. This level of parallelism is in addition to the function-level parallelism and the pixel-level parallelism previously mentioned. When there are multiple FPGA chips or when there are multiple memory ports per chip, the pixel-level parallelism may be explored in a different way. The image may be partitioned into “strips” of equal number of rows and each strip may be assigned to one single memory port (chip). When multiple masks are assigned to the same FPGA chip, there is an opportunity for masks to share hardware. The overlapping adder tree used in [9] is one such example.

4. Case Study

When mapping an application on reconfigurable systems, there are usually many design options by exploring parallelism at various levels. For parallelism at the pixel level different data buffering and allocation mechanisms require different amount of FPGA area, number of memory ports, and memory size which are all constrained by the FPGA co-processor board. As a result, the constraints can be used to prune the number of design options. This point is illustrated in this section by first implementing an infrared automatic target recognition (IR ATR) application on two different commercial FPGA boards and then using the data obtained from the implementation to estimate the resource requirements of different design options.

In the following subsections, the IR ATR algorithm and the two FPGA boards are first introduced. The design decision-making process is then described.

4.1. The IR ATR Algorithm

The IR ATR algorithm locates and identifies ground vehicles based on a single IR image frame. The IR ATR algorithm consists of many conceptually simple steps that each evaluates the matching between an image area and a template pair of target and background.

An overview of the algorithm is shown in Figure 8. The algorithm contains several steps, called Round 0, Round 1, Round 2, ..., and Round 5. The first step, Round 0, is applied to the whole image of size 480×640 to identify the location and the *target super-group* of *regions of interests* (ROIs). Here an ROI is an image pixel whose surrounding area meets a certain criterion and is considered a candidate for further investigation. Usually no more than 20% of the image pixels become ROIs. (For the four test images available to the authors, less than 5% of pixels are ROIs.) Six template pairs are used in Round 0.

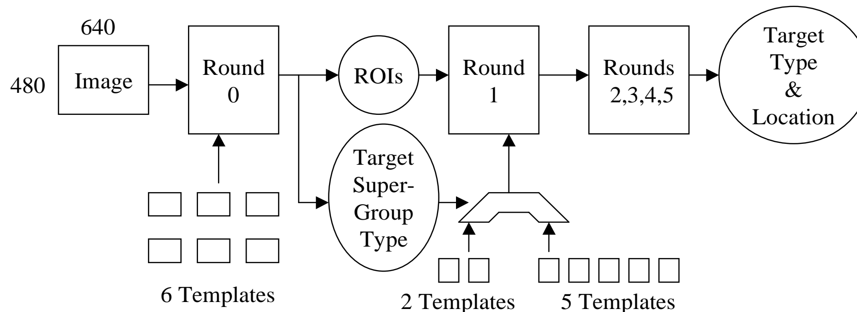

Figure 8. The ATR algorithm

Only ROIs are passed to the second step, Round 1, for further hypothesis testing and classification into *target groups*. Depending on which *target super-group* an ROI belongs to, Round 1 uses either two or five templates. ROIs that passed the Round

1 test are further tested through the remaining rounds and at the end, several pixels survive with their target types identified.

Because of the pruning process, Round 0 is computationally the most expensive step, followed by Round 1. Hence these two rounds were targeted for FPGA acceleration. They were first implemented on the G900 board (see [3]), and then ported to the StarFire FPGA board. But as will become clear in the section, "porting" to a different FPGA board may drastically change the constraints and necessitate the "invention" of a very different design, at least from a human designer's point of view.

4.2. Hardware Platforms

The first reconfigurable computing platform used in this application is a 180 MHz Pentium-pro personal computer hosting a G900 FPGA board which is a PCI bus based board manufactured by Giga Operations Corporation. This board consists of eight computing modules (XMODs) where each XMOD contains two XC4020E FPGA chips, 2 MB DRAM, and 256 KB SRAM. Each XC4020E chip on an XMOD is connected to a 128 KB SRAM through a 16-bit wide data port (see Figure 9). The host processor has to go through FPGA chips to access SRAM. Also, the host processor has to go through YFPGA to access XFGPA. In our implementation neither the XFPGA nor the DRAM is used so to reduce the design complexity. As a result, an XMOD in our case can be considered as a single XC4020E chip connected to a 128 KB SRAM through a 16-bit wide data port. While the FPGAs can run at two clock rates of 33MHz and 16MHz, the host and memory interfaces are limited to 16MHz.

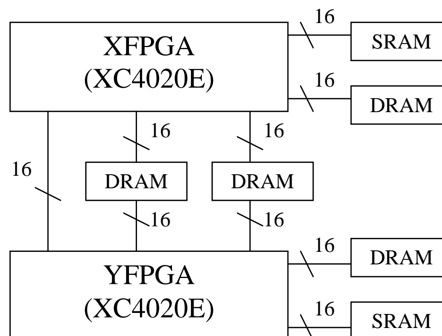

Figure 9. The XMOD architecture

The second reconfigurable platform used for this application is a 200 MHz Pentium-pro personal computer hosting a StarFire FPGA board which is a PCI bus based board manufactured by Annapolis Micro System, Inc. The board has one Xilinx Virtex XCV1000 chip as a processing element (PE) and two local 1MB SRAMs, labeled as "Left Mem" and "Right Mem" in Figure 10, each of which has a 32-bit

wide data port. The board has two mezzanine cards attached to it. Each mezzanine card contains two 1MB SRAMs, each with a crossbar and a 64-bit wide data port. The PE can access the SRAMs on mezzanine card through the crossbar. The Virtex FPGA on the board can run at clock up to 100MHz, but the PCI clock runs at 33MHz.

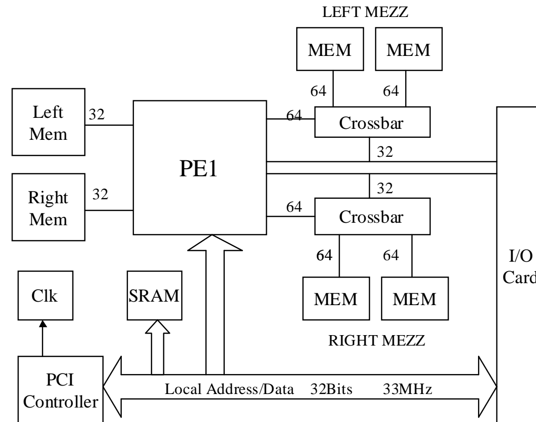

Figure 10. The StarFire architecture

4.3. Exploring Function-level Parallelism

The function-level parallelism depends on the particular computation performed on each pixel location with a template. Since this level of parallelism is not the focus of this section, it suffices to say that two basic FPGA computational blocks were implemented, one for Round 0 and the other for Round 1. (Please refer to [3] for details.)

The basic block for Round 0 corresponds to the computation of applying one template-pair to a single pixel location. The block reads in one pixel each clock cycle where each pixel is one byte long. It takes 60 clock cycles before the block can start applying the template-pair to the next pixel. (A template-pair contains 60 *active points*.) Note that no row buffering was used because it would have taken too much FPGA area. The basic block for Round 1 corresponds to the computation of applying one template to a single pixel location. The block consumes four pixel values from external memory each clock cycle and it takes 20 clock cycles before the block can start applying the template to the next pixel. (A Round 1 template contains 80 *active points*.)

4.4. Exploring Pixel-level Parallelism

To explore the pixel-level parallelism and therefore maximize the throughput, as many copies of those basic blocks as possible are to be squeezed into the FPGA

Table I. Area requirements for various Round 0 components

		G900 (XC4020E) Area (CLBs)	StarFire (XCV1000) Area (Slices)
One Basic Block		268	232
Internal Buffer	$k = 2$	72	-
	Hybrid($k = 4$)	-	114
	$k = 4$	-	129
FIFO (for active pixels)		32	32
Controller		186	338
Sub-block A		110	-
Sub-block B		158	-
Multiplexer		46	-

chip under the constraints of FPGA area, memory size, memory port width, and the number of memory ports.

4.4.1. *Design Decision Making for G900 Mapping*

On each XMOD for the G900 board, the SRAM connected to FPGA chip has a 16-bit wide data port. Since the Round 0 basic block requires one pixel per clock cycle, the maximal number of copies that can be used is two. For Round 1, the basic block requires reading four pixels each clock cycle, which cannot be effectively supported by this board without re-designing the basic block. As a result, only Round 0 is considered for G900 board discussion. To combine the two copies of Round 0 basic blocks, there are two main options:

- *Option 1*: Use internal buffer (as in Figure 4)
- *Option 2*: Duplicate data storage (as in Figure 5(a))

In order to evaluate these options, FPGA areas for several Round 0 design components are listed in Table I where the CLB/Slice numbers are produced by using Xilinx's place and route tool. It should be pointed out that the controller CLB/Slice areas are for two copies of basic blocks in the case of G900 and for four copies in the case of StarFire. The controller areas may certainly change when using a different strategy for combining basic blocks. However the change is assumed to be little. Also, the CONTROLLER area for the G900 column is more precise in that it includes the host interface and SRAM interface while the CONTROLLER area for the StarFire column does not.

Using Table I, the number of CLBs needed for option 1 is at least 826 ($= 2 \times 268+72+32+186$) while that for option 2 is at least 754 ($= 2\times 268+32+186$). Both numbers are more than 705, which is 90 percent of the CLB count on an XC4020E FPGA chip (which contains 784 CLBs). Here we are assuming that only up to 90 percent of the CLBs can be used by the design. (At least ten percent are assumed to be used for placement and routing overhead.) It seems that neither option is feasible!

Further study indicates that the basic block contains two sub-blocks, A and B, and by doubling the clock frequency of sub-block B and using a multiplexer, sub-block B can be "shared" (time-multiplexed) by both copies of basic blocks. (Sub-block A cannot be shared because it uses the 33 MHz clock already for memory data access while sub-block B originally uses the 16 MHz clock.) With the sharing of sub-block B, either option can save 112 ($= 158 - 46$) CLBs. Therefore option 1 needs at least 714 CLBs while option 2 needs 642 CLBs. Now both options can potentially be implemented. But of course option 2 has a better chance of success.

4.4.2. *Design Decision Making for StarFire Mapping*

For the StarFire board, there are four memory modules that are available to the PE, two with 32-bit wide data ports and two with 64-bit ones. However each port in our study is treated as a 32-bit one so to have a modular design for all memory modules which would then simplify the design complexity. For Round 0, since each basic block consumes only one byte per clock cycle, a 32-bit wide data port can support four copies of Round 0 basic blocks and a maximal of 16 copies can be mapped to the FPGA chip. For Round 1, because the template is only applied to Round 0 ROIs and therefore random data access is required, a 32-bit wide data port is treated as 8-bit wide (i.e., 24 bits are ignored). Since the Round 1 basic block consumes four bytes per clock cycle, one byte from each of four memories, only one copy is put into the FPGA.

Because the FPGA chip (XCV1000) contains 12288 slices, which are more than enough to accommodate 16 copies of Round 0 basic blocks, one copy of Round 1 basic block, and the extra components, the FPGA chip area is not the constraint that plays a critical role in the decision making process. Instead the size of each memory module is *the* constraint. (Note that one frame of the image is roughly 300KB.) For each memory module, we may consider the following three options.

- *Option 1*: Duplicate data storage (as in Figure 5(b))
- *Option 2*: Use the hybrid strategy (as in Figure 7)
- *Option 3*: Use internal Buffer (as in Figure 6)

Because we choose to use only 32-bit ports, the effective size of each mezzanine memory becomes 512KB. As a result, there is no room to store duplicated image data and that rules out options 1 and 2. A design as illustrated in Figure 11 has

been successfully implemented on the StarFire board runnig a 40 MHz clock where Option 3 is implemented in the block “Four_Round0”.

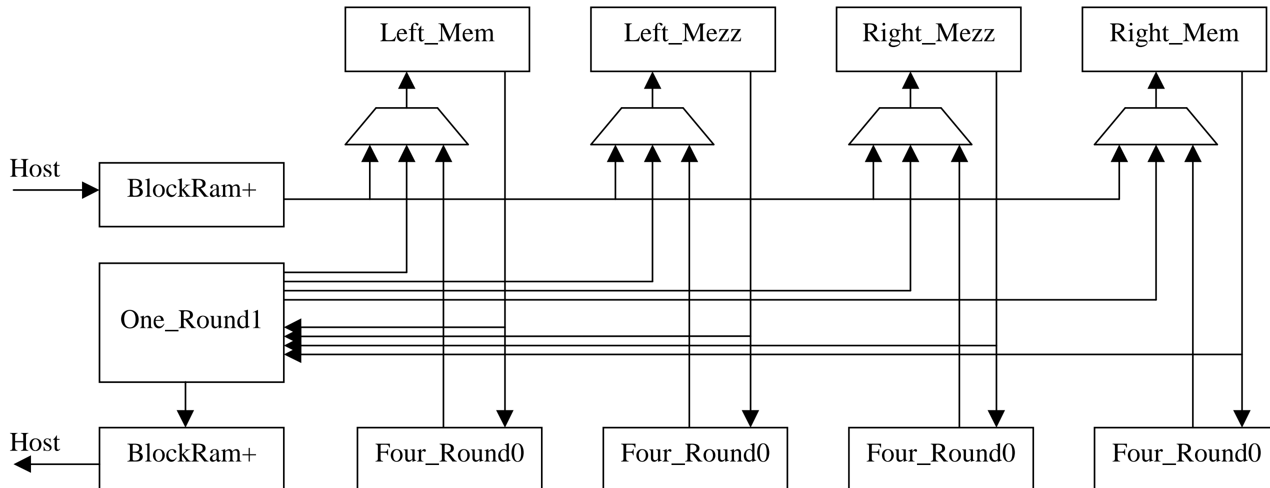

Figure 11. The design block diagram implemented in an XCV1000

Note that for G900 board, the SRAM size is only 256KB, not enough to hold the whole image frame. Therefore each frame is partitioned into a few overlapping strips of rows and the image is processed strip by strip. This is not considered in the case of the StarFire board to avoid the complexity in terms of handling striping for both Round 0 and Round 1. This is just one assumption that could have been removed. In this section, many such assumptions have been imposed so to reduce the number of options for the test case. Human designers are adept in making such assumptions to prune the design space.

5. Conclusions and Future Work

This paper is based on the work first published in the 1999 Parallel and Distributed Processing Techniques and Applications Conference [5]. It describes the *generalized template matching* (GTM) operation and characterizes the data allocation and buffering strategies for GTM operation on reconfigurable computers. The GTM operation offers ample opportunity in parallelization at different levels, including function-level, pixel-level, and multiple-mask-level. Several mechanisms that support different levels of parallelism are proposed and summarized in the paper. Such mechanisms were implemented on two commercial FPGA boards for an IR ATR application and design tradeoff were discussed.

Given a cost function that specifies the area-time tradeoff and constraints on FPGA areas, an optimal design depends on factors such as (1) the input image

size, (2) size of templates, (3) whether templates are constants or variables, and in the case of constants, the specific numeric values of templates, (4) computational operators in the generalized correlation/convolution, (5) numeric precision of the operators, and (6) data distribution (on memory or through bus). The problem is complicated for human designers because *hardware sharing* may be possible among multiple templates and *hardware reuse* may be necessary due to FPGA area constraints. Previously researchers have tried to produce good FPGA designs for special cases of the GTM operation with an ad hoc approach. It is more desirable to have systematic approaches that either enumerate and evaluate the design options or formulate the design problem as an optimization problem. Such approaches would enable the development of a parameterized generator that automatically generates FPGA designs given a set of user specifications for a GTM.

References

1. M. Alderight, E.L. Gummati, V. Piuri, and G.R. Sechi, “A FPGA-based Implementation of a Fault-Tolerant Neural Architecture for Photon Identification,” in Proc. of ACM/SIGDA International Symposium on FPGAs, pp. 166-172, 1997.
2. R. Cook, J.S.N. Jean, J.S. Chen, “Accelerating MPEG-2 Encoder Utilizing Reconfigurable Computing”, CERC/VIUF/IEEE Computer Society Workshop on “21st Century Electronic Systems Design: Breakthroughs in Quality and Productivity”, University of Dayton, December 1997.
3. J.S.N. Jean, X. Liang, B. Drozd, K. Tomko and Y. Wang “Automatic Target Recognition with Dynamic Reconfiguration,” to appear in the Journal of VLSI Signal Processing-System for Signal, Image, and Video Technology.
4. J.S.N. Jean, X. Liang, B. Drozd, and K. Tomko, “Accelerating An IR Automatic Target Recognition with FPGAs,” in the Proc. of IEEE Symposium on FPGAs for Custom Computing Machines, April 1999.
5. J.S.N. Jean, X. Liang, and K. Tomko, “Data Buffering and Allocation in Mapping Generalized Template Matching on Reconfigurable Systems,” in the Proc. of Parallel and Distributed Processing Techniques and Applications Conference, pp. 1111–1117, June 1999.
6. W.E. King, T.H. Drayer, R.W. Conners, and P. Araman, “Using MORRPH in an Industrial Machine Vision System,” in IEEE Symposium on FPGA Custom Computing Machines, pp. 18-26, 1996.
7. S. Singh and R. Slous, “Accelerating Adobe Photoshop with the Reconfigurable Logic,” in IEEE Symposium on FPGA Custom Computing Machines, pp. 18-26, 1998.
8. M. Shand and L. Moll, “Hardware/Software Integration in Solar Polarimetry,” in IEEE Symposium on FPGA Custom Computing Machines, pp. 18-26, 1998.
9. J. Villasenor, B. Schoner, K. Chia, C. Zapata, H. Kim, C. Jones, S. Lansing, and B. Mangione-Smith, “Configurable Computing Solutions for Automatic Target Recognition,” in IEEE Symposium on FPGA Custom Computing Machines, pp. 70–79, 1996.
10. C. Thibeault and G. Begin “A Scan-Based Configurable, Programmable, and Scalable Architecture for Sliding Window-Based Operations,” in IEEE TRANSACTIONS ON COMPUTERS, pp. 615-627, 1999.
11. M. Rencher, and B. L. Hutchings, “Automated Target Recognition on Splash 2,” in IEEE Symposium on FPGA Custom Computing Machines, pp. 192-200, April 1997.