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

Jack Jean, Xuejun Liang, 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 GTM operation on reconfigurable computers. Several mechanisms that support different levels of parallelism are proposed and summarized in the paper.*

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)[3, 7], neural networks[1], Adobe Photoshop[5], Solar Polarimetry[6], and machine vision[4].

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.

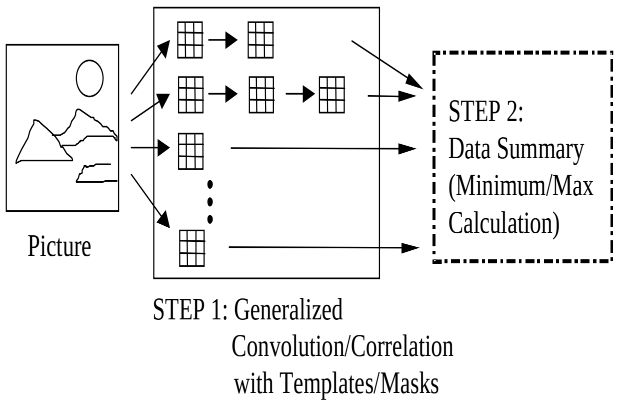


Figure 1: Generalized template matching

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 computation associated with each mask can simply be a prespecified 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[3, 7].

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].

This paper characterizes the mapping of the GTM operation on reconfigurable computers in terms of data buffering and allocation. 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.

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 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 pixels 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

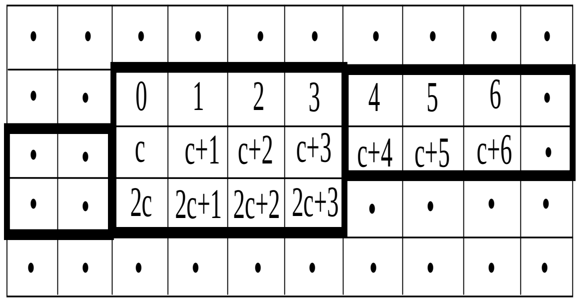


Figure 2: Pixels that need to be buffered for a 3×4 mask

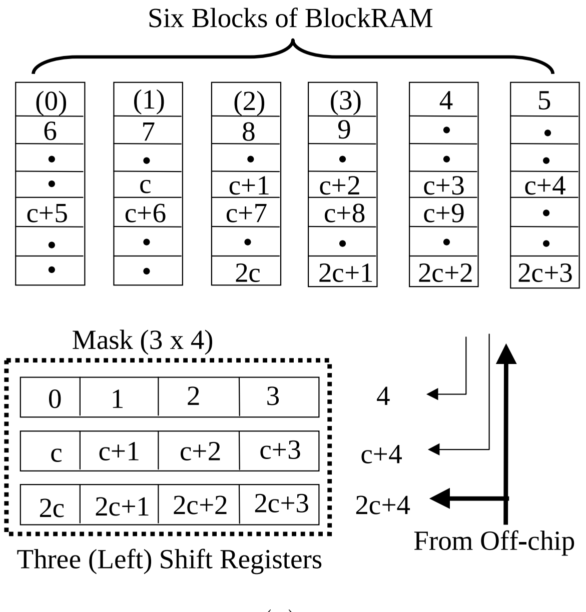
Figure 2 shows a 3×4 mask being applied to an image where the mask covers twelve pixels, 0, 1, 2, 3, c, $c+1$, $c+2$, $c+3$, $2c$, $2c+1$, $2c+2$, and $2c+3$. If the pixels enclosed by the bold lines are buffered inside the FPGA chip, moving the mask one pixel right requires only the reading of one extra pixel, i.e., pixel $2c+4$, from the external memory (or the host processor). 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 needs to be read exactly once. That would greatly reduce the memory bandwidth requirement.

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 [7] is an example.

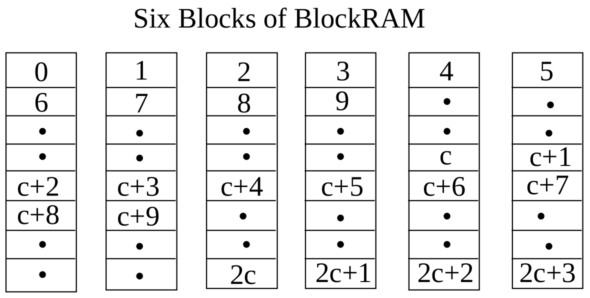
FPGA Buffers On Xilinx FPGA chips, there are three different mechanisms that can be used to buffer image rows.

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[7].
2. Use the RAMs that are based on function generators in CLBs. One CLB can store 32 bits without parallelism in accessing. 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).



(a)



(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 [7] for an example).

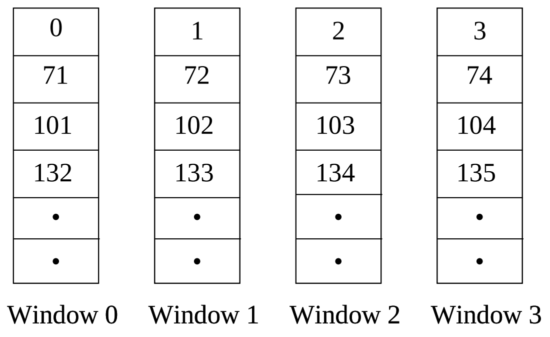
2.2 No Buffering of Image Rows

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.

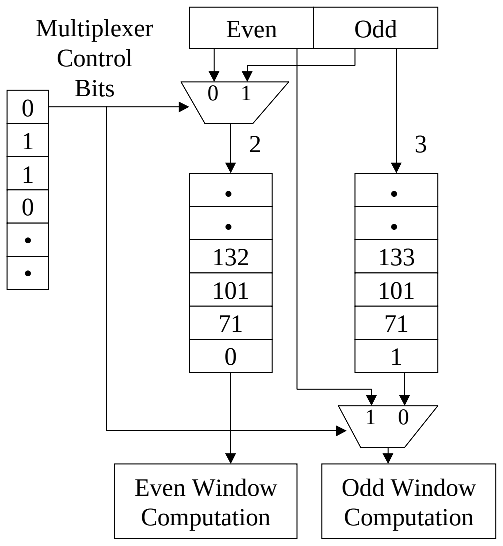
Pixel-Level Parallelism

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.

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 pixel-level parallelism so that windows 0 and 1 can be evaluated in parallel, there is a need to provide simultaneously a pair of consecutive pixels n and $n+1$ to those two copies of computation units and n is not always even or always odd. (More specifically



(a)



(b)

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

the values of n are 0, 71, 101, 132, and so on.) The ability to provide simultaneously a pair of consecutive pairs can be achieved by using either a redundant external data storage or a small internal buffer.

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. 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 [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.

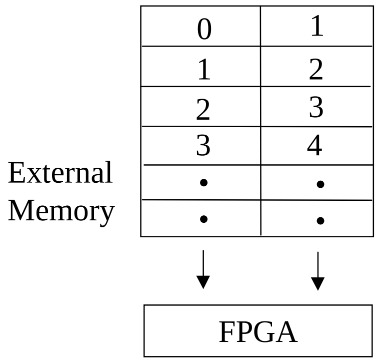


Figure 5: Using redundant external data storage ($k = 2$)

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. It is also possible to use a hybrid scheme that mixes the two schemes. An example when $k = 4$ is shown in Figure 6.

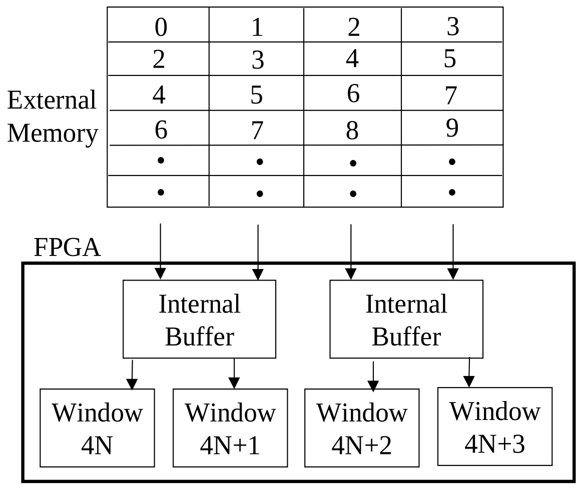


Figure 6: Using a hybrid scheme ($k = 4$) with redundant external data storage and small internal buffers

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.

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 [7] is one such example.

4 Conclusions and Future Work

This paper 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.

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 systematical 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.

Acknowledgments

This research was supported by DARPA under Air Force contract number F33615-97-1-1148, an Ohio State investment fund, and an Ohio State research challenge grant. Xilinx Inc. donated an FPGA design tool and FPGA chips on XMODs.

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, and K. Tomko, "Accelerating An IR Automatic Target Recognition Application with FPGAs," to appear in IEEE Symposium on FPGA Custom Computing Machines, April 1999.

[4] 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.

[5] S. Singh and R. Slous, "Accelerating Adobe Photoshop with the Reconfigurable Logic," in IEEE Symposium on FPGA Custom Computing Machines, pp. 18-26, 1998.

[6] M. Shand and L. Moll, "Hardware/Software Integration in Solar Polarimetry," in IEEE Symposium on FPGA Custom Computing Machines, pp. 18-26, 1998.

[7] 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.